GENERAL SYMMETRIC DISTRIBUTED TERMINATION DETECTION

R.B.Tan and J.van Leeuwen

RUU-CS-86-2
January 1986

Rijksuniversiteit Utrecht

Vakgroep informatica

Budapestlaan 6 3584 CD Utrecht
Corr. adres: Postbus 80.012 3508 TA Utrecht
Telefoon 030-53 1454
The Netherlands

GENERAL SYMMETRIC DISTRIBUTED TERMINATION DETECTION

R.B.Tan and J.van Leeuwen

Technical Report RUU-CS-86-2
January 1986

Department of Computer Science
University of Utrecht
P.O.Box 80.012, 3508 TA Utrecht
the Netherlands

GENERAL SYMMETRIC DISTRIBUTED TERMINATION DETECTION*

Richard B.Tan** and Jan van Leeuwen

Department of Computer Science, University of Utrecht
P.O.Box 80.012, 3508 TA Utrecht, the Netherlands

Abstract. Several distributed algorithms for termination detection in processor networks are presented that are more general than the previous algorithms for this problem. In particular the solutions presented here are "general" in the sense that they are fully asynchronous and make no use of global knowledge, and "symmetric" in the sense that processors are only distinguished by a unique processor id and no processor acts as a preassigned leader. Essentially the distributed algorithms are obtained by superimposing termination detection on a suitable distributed election algorithm. Efficient solutions to the distributed termination detection problem are presented when the underlying processor or control network is a ring, a tree, or a general network.

1. Introduction. Termination detection is an important control task in distributed systems. Consider a network of N processors cooperating in a distributed computation. (Alternatively, consider a concurrent system of N processes.) Each processor is either busily performing a computation and is "active", or it has finished its current task and is idle ("passive"). An active processor can communicate with any of its neighbors by sending it a message, typically containing a new task for it. Upon receipt of a message an active processor simply remains active (presumably now for a "longer time") but an idle processor changes mode

---

*This work was carried out while the first author visited the University of Utrecht, supported by a grant from the Netherlands Organisation for the Advancement of Pure Research (ZWO).

**Author's address: Dept. of Mathematics and Computer Science, University of Sciences & Arts of Oklahoma, Chickasha, OK 73018, USA.

and becomes active also. After finishing its current tasks an active processor becomes idle again. When the entire distributed computation is done, all processors are necessarily idle. "Distributed termination" requires that the condition of global idleness is detected in a distributed manner, in order for the processors to finish (terminate) the computation.

Distributed termination detection is by no means a new or original problem. Several solutions to it have been proposed in the recent literature, under varying assumptions about the underlying model of distributed computation. A distributed termination detection algorithm (or DTDA, for short) must satisfy the following requirements (cf.Apt[1]):

(a) it will neither deadlock nor cause deadlock,

(b) if the distributed computation has finished, the DTDA will eventually detect it,

(c) the algorithm will not falsely detect termination, and

(d) if the distributed computation has not finished, then eventually another action of it will be executed.

Thus, a DTDA must be a protocol that runs in the "session layer" of the network control software (see e.g. Tanenbaum [29]) without essentially interfering with or blocking a distributed computation by the network. The design of correct DTDAs is a non-trivial problem, and most solutions known to date make one or more simplifying assumptions about the underlying network. The following types of restrictions are often encountered:

(i) a leader node (a "root") is assumed that initiates and orchestrates the distributed termination detection,

(ii) a special (sub-)topology is assumed to route control messages in a desired way, viz. a ring or hamiltonian circuit,

(iii) global knowledge is assumed, like the number of processors or the diameter of the network.

Figure 1 summarizes many of the known solutions to the distributed termination detection problem and the assumptions that are used. In this paper we present solutions that are completely general, in the sense that none of the above restrictions are assumed.

DTDAs that assume no leader and/or no special (sub-)topology have

algorithm	leader node	special subtopology	global knowledge
Sintzoff (1978)	+	hamilt.cycle	
Dijkstra (1978)	+	-	
Dijkstra & Scholten (1980)	+	-	
Francez (1980)	+	spanning tree	
Francez, Rodeh, & Sintzoff (1981)	+	hamilt.cycle	
Gouda (1981)	+	ring	
Francez & Rodeh (1982)	+	spanning tree	
Cohen & Lehmann (1982)	+	spanning tree	
Misra & Chandy (1982)	+	-	
Dijkstra,Feijen,& van Gasteren(1983)	+	ring	
Misra (1983)			
Rana (1983)	-	hamilt.cycle	+
Lozinskii (1984)			
Topor (1984)	+	spanning tree	
Apt & Richier (1984)	-	ring	+
Richier (1984)	-	ring	+
Eriksen & Skyum (1985)	-	-	+
Szymanski, Shi,& Prywes (1985)	-	-	+
Chandy & Lamport (1985)	-	-	(overhead)
this paper (1986)	-	-	-

Figure 1 Some distributed termination detection algorithms and the assumptions they use.

received some attention recently. Richier [26] presents a solution that requires only O(logN) bits of local storage per processor and proves this to be minimal, still assuming a circular arrangement of the processors. Eriksen & Skyum [11] present the most general "symmetric" solution for arbitrary strongly connected networks, assuming only the diameter of the network as global knowledge in every processor. A rather similar DTDA of Szymanski, Shi, & Prywes [28] gets around the latter restriction by including a distributed algorithm to determine the network diameter. An entirely different, but completely general and symmetric DTDA follows by applying the "global state detection" algorithm of Chandy & Lamport [5], although this algorithm requires a costly state-sampling routine and is not geared to this particular application. (See Bougé [4] for an efficient repetitive version of the Chandy-Lamport algorithm.) The solutions we present only make the (usual) assumption that processors are distinguished by unique but otherwise arbitrary identification numbers, and require no further special assumptions or asymmetry and do not carry much overhead. (In the case of a tree network no identification numbers are needed at all, in fact.)

The general symmetric DTDAs we present are based on the combination of termination detection and distributed "election". Recall that distributed election (leader finding, symmetry breaking) algorithms are protocols for determining a leader node in symmetric networks. It would be all too simple and indeed be very inefficient to solve the distributed termination detection problem by first running a distributed election algorithm and then switching to any of the existing DTDAs that require the control by a leader node. The two "passes" delay the detection in case of fast termination and presumably lead to a higher message-complexity for the problem than is required. Intuitively termination detection can be superimposed on the distributed election protocol and the two tasks viewed as one. In this paper we will make this intuition precise and prove a number of general symmetric DTDAs obtained from this paradigm. We assume familiarity with some distributed election algorithms (see e.g. Bodlaender & van Leeuwen [3], Chang & Roberts [6], or Hirschberg [19]) and use the "black-white" paradigm as found in several recent non-symmetric DTDAs (see e.g. Dijkstra, Feijen, & van

Gasteren [9], or Topor [30]). In sections 2,3, and 4 we present efficient general symmetric DTDAs for ring, tree, and general (sub-)topologies, respectively.

2. A symmetric distributed termination detection algorithm using a ring sub-topology. Consider a distributed (asynchronous and connected) network of N processors distinguished by unique, but otherwise arbitrary identification numbers. Let each processor have buffer queues of sufficient capacity associated with its links so it can send or receive messages at will and messages are always received in the order in which they were sent (over a link). We assume that the communication subsystem is error- and failure-free.

In this section we present a solution to the symmetric distributed termination detection problem for the case that the processor network contains a (unidirectional) ring or hamiltonian cycle as a distinguished sub-topology. The idea is that the termination detection protocol is performed over the ring only. It follows that there can be two types of messages in the network at any moment in time:

(a) activation messages, which are generated and sent by active processors in the course of the distributed application and which convey (new) computational tasks to other active or idle processors,

(b) token messages ("termination probes"), which are generated and sent by all processors as prescribed by the termination detection algorithm.

Activation messages can be sent to any neighboring processor (i.e., over any existing link), but token messages can only be sent to the neighboring processor on the predefined ring.

The basic idea is to derive a symmetric DTDA by modifying any of the existing distributed election algorithms for rings. We will use the election algorithm due to Chang & Roberts [6], because the algorithm is fast and requires no particular auxiliary registers in the processors. (Recall that the basic algorithm of Chang & Roberts aims at identifying the processor with the largest id as the leader node.) Consequently idle processors can be in one of two states, idle-black or idle-white, with the following intuitive meaning:

(i) a processor is idle-black if it has "recently" turned idle and no token message from a (candidate) leader passed it since, and

(ii) a processor is idle-white if it is idle and a token message from a (candidate) leader passed by to record it (and hence, in particular, the processor is not a leader-candidate itself).

It is important to realize that the DTDA we present does not explicitly compute the leader node, despite the fact that we use the terminology. Initially processors are active or idle-black.

Distributed termination detection is accomplished through a regular exchange of token messages over the ring. Roughly speaking, token messages will only be generated when it is reasonable to do so, i.e., when an active processor is done with its current tasks and switches to idle-black. In order to distinguish between successful and unsuccessful termination probes, token messages can be black or white also (cf.[9]), with the following intuition behind it:

(i) a token message is white if it is (still) reasonable to suspect termination and the processor that originally sent it is (still) regarded as the leader, and

(ii) a token message is black if it is reasonable to suspect that there is no termination yet.

We note that white token messages will be purged (destroyed) when it is determined that the sending processor is not the leader, i.e., as soon as it is defeated by a processor with a larger id. Token messages will be of the form <p,c> with p the id of the originator of the message and c the color of the message. The color of a token message can change as it travels over the ring according to the rules of the protocol.

We will now give a description of the complete DTDA in "protocol R" below. Different actions can take place depending on the state of a processor (active, idle-black, or idle-white) and, hence, the specification follows the distinction by state.

Protocol R

{The actions are specified for an arbitrary processor p. It is assumed that p knows its neighbors in the predefined sub-topology (the ring), viz. over which link it must send token-messages and over which link it

receives them. Its actions are described as "rules" depending on its state.}

Active state:

{A processor in this state is actively computing on subtasks of the distributed application. It can send activation messages to any neighboring processor.}

Rule I

While computing, store incoming messages (both activation and token messages) in the buffer queues. Upon finishing the basic computational task, **goto** idle-black.

Idle-black state:

{A processor in this state has "recently" turned idle. It can be reactivated through activation messages (from active processors) only.}

Rule II

Repeat the following action until a termination signal comes in or a change of state occurs, meanwhile storing incoming messages (if any) in the buffer queues.

```
if there is an activation message in any buffer queue then
   begin
      delete it from the queue and initialize the processor for the
      task; goto active
   end
else
    {attend to token messages, if any have come in..}
    begin
       delete all token messages from the queue and store them
       in a set S;
    if S ≠ Ø then
       {some processors preceding p have started a termination detection
       probe..}
       begin
          let q be the maximum id of a message in S;
          case "test as indicated" of
          q>p: begin
```

```
                send token message <q, black>; ----------------(***)
                goto idle-white
            end;
        q=p: if <p,black>ε S then send token message
            <p,white> else send termination signal around the
            ring; ----------------------------------------(**)
        q<p: if no token message was ever sent by p before
            then send token message <p,white> ---------------(*)
        end
    end
  else
    {no processor preceding p apparently started a termination
    detection probe, so p better make sure and start one..}
    if no token message was ever sent by p before then
    send token message <p,white>;-------------------------------(*)
  S:=Ø
end;
```

Idle-white state:

{A processor in this state is idle, and a termination detection probe passed by from a (candidate) leader. It can be reactivated through activation messages (from active processors) only.}

Rule III

Repeat the following action until a termination signal comes in or a change of state occurs, meanwhile storing incoming messages (if any) in the buffer queues.

```
if there is an activation message in any buffer queue
then
   begin
      delete it from the queue and initialize the processor for the
      task;goto active
   end
else
   {attend to token messages, if any have come in..}
   begin
     delete all token messages from the queue and store them
```

```
        in a set S;
        if S≠∅ then
           {some processors preceding p have started a termination
           detection probe..}
           begin
               let q be the maximum id of a message in S, and c the
               color of the corresponding token message;
               if q>p then send token message <q,c>
           end
         else
           skip;
         S:=∅
    end;
```

Observe that activation messages are always given first priority in any idle state. (Actually, the protocol should have been written so the arrival of an activation message acts as an interrupt in the idle state.) Observe also that it is not necessary to have an explicit "set" S of token messages at the particular points in the protocol: just keeping the token message with the largest id is sufficient, because this is all we use from it. We will discuss another optimization of protocol R at the end of this section.

The description of the protocol should be fairly self-explanatory. If a processor in the idle-black state receives "somebody else's" termination detection probe, it blackens the probe (because it probably was active when the probe was initiated and may have reactivated other processors that already were in some idle state). If the processor is a valid candidate for leadership, it takes over the initiative and starts a new (white) termination detection probe. Otherwise it just passes the blackened token message on, until somebody (e.g. the original sender) will take the lead to begin a new termination detection probe. (Note that this is the essential symmetry-breaking part of the protocol.) A processor can only move to the idle-white state if it receives a higher-value token in the idle-black state. It thus gives way to the termination detection probes of candidate leaders, and merely acts as a relay station for incoming token message (unless it turns active again).

In particular, no processor in the idle-white state can issue a new probe.

Lemma 2.1

(i) In protocol R there can be no two token messages with the same id on the ring at any one time.

(ii) Only the leader node (i.e., the processor with largest id) can possibly receive its own token message back.

(iii) The leader node can never turn idle-white.

Proof.

(i) This property is guaranteed by the lines (*) in protocol R. All other actions in the protocol do not involve the generation of new messages.

(ii) This follows by inspecting Rule II. Note that any token message <q,c> is eventually stopped and destroyed in a processor with an id larger than q, unless q is the maximum id.

(iii) This follows by inspecting Rule II as well. Only token messages <q,c> with q>p can put p into the idle-white state, and this can never happen when p is maximum. □

As a consequence of lemma 2.1. (ii) we note that line (**) of protocol R can only be executed by the leader node. (Subsequently any processor p that executes this statement knows that it currently is the leader!)

We now prove the correctness of protocol R. From the description of the protocol it is clear that it cannot deadlock and that it never stands in the way of the basic distributed computation. Thus we need only consider the termination detection capabilities of the protocol.

Lemma 2.2. If all processors are idle, then protocol R will detect it and cause termination.

Proof.

Let all processors be idle (which means that some will be idle-black and some perhaps idle-white). By lemma 2.1. (iii) the processor p with largest id (the "leader") must be idle-black. By inspecting Rule II it is clear that p must have generated a token message <p,white> the first

time it turned idle-black (including the case where it started in the idle-black state). This token message may have been blackened along the ring, but it is never killed and (hence) it is eventually delivered at p again. Consequently, p either generates a <p,white> message spontaneously (because it is the first time it turned idle-black and executes line (*)) or eventually it receives the token message back that it sent out earlier. If a token message <p,white> is generated now, it will be sent around the ring and be passed on by all idle processors. Idle-white processors will pass on the message without changing its color, and idle-black processors will blacken it according to line (***) of the protocol but now turn idle-white as well. If p receives a token <p,black> back, the next white token message it generates will see only idle-white processors on the ring (except, of course, p itself). Thus p will eventually receive a token message <p,white> and (hence) generate a termination signal according to line (**) of the protocol. □

Lemma 2.3. No processor can terminate unless all other processors are idle or terminated.

Proof.

A processor can terminate only when (it is idle and) it receives the termination signal. By inspecting Rule II and lemma 2.1.(ii) it is clear that the termination signal can only be generated by the processor p with largest id (the "leader") and that p will generate the signal only when its token message has made a full tour around the ring without changing color and p is idle itself. Consider the moment that p is idle (hence, idle- black) and generates the white token message M≡<p,white> that makes the full tour without changing color. Suppose there is any processor q≠p on the ring that is still active. Consider M travelling towards q. If M arrives at q while it is still active or idle-black, M will be blackened by q according to line (***) of the protocol. On the other hand, q can be idle-white only if a token message of an id q' with q'>q has passed it before M got there. But q' must be between p and q, and it can generate a token message only if it turned idle sometime before q did (and clearly, by the ordering of the token messages, before M arrived at q'). M must pass q' and thus two cases can arise: either

q' is (still) idle-black when M arrives and will blacken M by line (***) of the protocol, or q' has turned idle-white. The latter case, again, can occur only when the token message of some processor q" with q">q' passed it before M did, where q" is between p and q' and q" turned idle before q' did and, by the ordering of the token messages, before M arrived at q". By repeating this argument it is clear that M must be caught by a processor in the idle-black state, some time after it was sent out by p. Thus, if there was (still) an active processor on the ring the very moment M was sent out, it would not have made its full tour around the ring without eventually turning black. Consequently the termination signal is generated only when all processors are no longer active. □

We conclude:

Theorem 2.4. Protocol R is a correct symmetric distributed termination detection algorithm (for networks with a ring sub-topology).

As to the message-complexity of protocol R we observe that in principle every processor can start a termination detection probe the first time it switches to the idle-black state. In fact, every processor will do so unless it has a higher-value token message waiting in the buffer queue when it turns idle-black. (It will subsequently blacken this token message, pass it on and switch to the idle-white state.) As a processor cannot possibly be the leader when this happens, Rule II can be further optimized by prohibiting every processor that has ever been idle-white from generating its token message (when it turns idle-black and would have the chance to do so). By inspecting the protocol it should be clear that generated token messages travel very much like in the Chang & Roberts algorithm, when they are not upheld by active processors. It is well-known that the latter algorithm requires O(NlogN) message transfers on the average until the leader node, viz. its token message, is the sole "survivor" ([6], see also [3]). Note that the required message transfers never stand in the way of the basic distributed computation and run to completion only when sufficiently many state changes have ocurred around the ring. In this sense, protocol R

ultimately converges to the detection algorithm of Dijkstra, Feijen, & van Gasteren[9].

One further optimalisation of Rule II can drastically reduce the message-complexity of protocol R under certain circumstances. (A similar observation for distributed election algorithms was first reported by Kutten[20].) If a processor turns idle-black for the first time and it finds token messages waiting in its buffer queue, then it can decide to leave the initiative of termination detection to the other processors and simply not to enter the run for leadership (regardless of its id). Effectively it means that the processor acts as if its id is "smaller" than any id in a current token message and that, by an earlier optimalisation, it is prohibited from ever initiating a termination detection probe of its own. With this optimalisation the (implicit) leadership is settled among the "early finishers", which can be very efficient. The optimalisation heavily relies on the symmetric distributed character of the protocol.

3. A symmetric distributed termination detection algorithm using a spanning tree sub-topology. Consider again a distributed (asynchronous and connected) network of N processors. We make very much the same computational assumptions about the processors as before, except that in this section we do not require that the processors are distinguished by unique id's. However we do assume that a processor is able to distinguish its I/O ports internally and (hence) that for each message that it receives it knows the link over which the message arrived. We assume again that the communication sub-system is error- and failure-free.

In this section we present a solution to the symmetric distributed termination detection problem for the case that an arbitrary spanning tree of the network is fixed as a distinguished sub-topology for the purposes of the termination detection protocol. It is assumed that every processor knows which of its ports correspond to links of the tree. As the constraint for this case is easily met in any network without adding extra communication lines, it has traditionally been considered as the most general version of the distributed termination detection problem (see e.g. figure 1 for references). The feature that

distinguishes our solution from previous DTDAs for the problem is that we do not assume that a root is given for the spanning tree. The (symmetric) DTDA we present will not even "ultimately converge" to the case where a particular node is established as a leader (or root), unlike the protocol in section 2. It is best to think of the symmetric DTDA as using a "floating" root, although the details are a little more tedious. The degree of a node p in the spanning tree will be denoted by deg(p).

We recall that while the basic distributed computation is being performed there can be two types of messages in the network: activation messages, which can be sent by active processors to any neighbor (i.e., over an existing link), and token messages, which can only be sent by idle processors over links of the spanning tree. As before, token messages can be black or white. Because we do not assume distinguishing id's for the processors, token messages simply have the form <c> with c some color. It is convenient to view the colors as Boolean values, with black acting as "0" and white as "1". (Thus conjunctions $c_1 \wedge .. \wedge c_n$ will always be black as long as there is one c_i, $1 \leq i \leq n$, that is black.) For the purposes of the symmetric DTDA we actually require a third type of control message, called a *-message (or: a "repeat" message). The symmetric DTDA will be intimately related to the termination detection algorithm of Topor [30], but without the reliance on a root node of the latter.

The basic idea is to derive a symmetric DTDA by imposing a termination detection regime on a suitable distributed election algorithm for trees. We will use the election algorithm due to Hirschberg[19] because it is fast and simple. However, rather than using it to "ultimately converge" to a leader node (which we can't do anyway because processors have no id's), we use its "logic" to perform repeated bottom-up termination detection probes and to know when one probe has reached "the root" (whatever it will turn out to be) and the next one can be started (if needed). The termination detection probes proceed very much like in Topor's algorithm [30] (from the leaves towards "the root") and are implemented by "wavefronts" of token messages. An interesting aspect of the algorithm is that every termination detection probe can come up with its own "root" (depending on the actions and speeds of the processors)

and hence that this node may be a different one for every wave. There can be at most one wavefront in progress at any one time. Depending on their situation, processors can again be "active" or "idle". As in section 2, idle processors can be in one of two states: idle-black or idle-white (with a similar intuitive meaning). Initially, processors are either active or idle-black.

We will give a detailed description of the complete symmetric DTDA as "protocol T" below. The main difficulty will be to know when a wavefront has converged to a single node, whose subsequent task it is to either call for a repeat (by sending out *-messages) or conclude termination. Normally convergence is concluded by a processor p when it has received token messages from the current wave over all incoming tree-links (and p considers itself the "current leader" when this happens). The following complication can arise. The normal operation of the protocol requires that if a processor p has received deg(p)-1 token messages (and is idle), then it must advance the wavefront by sending a token message to the one neighbor q from which no token message was received. But if q was in a similar situation as p and (consequently) decided to send its token message to p more or less at the same moment, then we end up having no leader at all unless processors carefully watch out for this situation. It is clear that in this case both p and q become leaders and we cannot choose between them (the "double leader" case). For the moment we will avoid the possibility of double leaders by requiring that token messages never bypass each other in a link. At the end of this section we discuss the necessary modifications of the protocol in order to handle the general case as well.

Protocol T

{The actions are specified for an arbitrary processor p. Recall that processors have no id's, and that we assume here that token messages do not bypass each other in a link. Each processor knows which of its links belong to the pre-defined spanning tree of the network.}

Active state:

{A processor in this state is actively computing on subtasks of

the distributed application. It can send activation messages to any neighboring processor.}

Rule I -

While computing, store incoming messages (of any sort) in the buffer queue of the link over which they come in. Upon finishing the basic computational task, **goto** idle-black.

Idle-X state, Xε{black,white}:

{A processor in this state is idle. It can be reactivated through activation messages (from active processors) only. X is the "color" of p. Note that repetition of the code below is not implicit and must be triggered by the receipt of a *-message, even after turning active and idle-black again.}

Rule II - X

{p acts according to the instructions below. Instruction o has top priority and pre-empts all other instructions when it can be applied.}

o. Store incoming messages (of any sort) in the corresponding buffer queue. If there is an activation message in any queue, then delete it from the queue, initialize p for the appropriate task, and **goto** active.

1. When token messages $\langle c_1\rangle,\ldots,\langle c_l\rangle$ (l=deg(p)-1) have been received from deg(p)-1 different neighbors (in the spanning tree), delete them from the corresponding buffer queues and send a token message $\langle c\rangle$ with $c=c_1\wedge\ldots\wedge c_l\wedge X$ to the one remaining neighbor on the spanning tree (from which no token message was received yet).

2a. After a token message is sent as specified under 1, p goes to idle-white ("X: =white").

3a. When token messages $\langle c_1\rangle,\ldots,\langle c_{l+1}\rangle$ (1+l=deg(p)) have been received from all different neighbors in the tree, delete them and do the following. If $c_1\wedge\ldots\wedge c_{l+1}\wedge X$ = white then send a termination signal to all processors, otherwise send a *-message to every neighbor in the spanning tree and go to idle-white ("X:=white").

4. When a *-message is received from any neighbor q in the

spanning tree and deg(p)>1, then delete it from the corresponding queue and send ("pass on") the *-message to the deg(p)-1 neighbors ≠q in the spanning tree.

5. When a *-message is received from a neighbor in the spanning tree and deg(p)=1 ("p is a leaf"), then delete it from the corresponding buffer queue and repeat Rule II-X.

Observe that activation messages are always given first priority, in the sense that they interrupt a processor in Rule II and send it back to active. Instructions themselves are considered atomic.

The description of the protocol is fairly self-explanatory. By inspecting Rule II (viz. instructions 1 and 5) it should be clear that only idle leaves of the spanning tree can initially send out token messages and start up a new termination detection probe (a "wave"). A new wave cannot start until the preceding one has finished, by virtue of instruction 3 and the fact that no leaf will begin Rule II unless it has received another *-message. (For the initial condition we may assume that all leaves start as if they received a *-message.) Note that by assumption there can be only one processor p during the current wave where, eventually, instruction 3a can be applied, although we can't say in advance which processor it will be. If instruction 3a is applied in processor p, p essentially acts as a leader, ends the current wave, and decides to either call for another wave or conclude termination. Observe that a (new) wavefront need not start at all leaves simultaneously.

We now prove the correctness of protocol T. From the description of the protocol it is clear that it cannot deadlock and that it never stands in the way of the basic distributed computation. Thus we need only consider the termination detection capabilities of the protocol.

Lemma 3.1. If all processors are idle, then protocol T will detect it and cause termination.

Proof.

Let all processors be idle, which means that some will be idle-black and other are perhaps idle-white. When the leaf-processors of the

spanning tree turned idle (hence, idle-black) for the first time (or, when they started idle), they will have initiated a first termination detection wave which, subsequently, keeps the termination detection process going. Consider the current wavefront, at the moment the last processor turned idle. As there are no active processors anymore that can hold it up, the current wave will run to completion and end (by assumption) in some processor p that executes instruction 3a. If p concludes "white", then it will generate a termination signal and correctly decide termination. Otherwise p will turn white and send *-messages out which, by instruction 4, eventually reach all leaves of the spanning tree and trigger a new termination detection wave. As all processors are idle, the wave will propagate unhindered and whiten every idle-black processor it still encounters. Eventually the wave will end at some processor p' that now acts very much like p before. If p' cannot conclude termination now, then the next wave that it triggers will find all processors idle-white and thus converges at some processor p" with all colors white. Hence, either p' or p" concludes termination. □

Lemma 3.2. No processor can terminate unless all other processors are idle or terminated.

Proof.

(The result is similar to Topor's [30] but we have to follow a different proof technique.) Given a current wavefront W, any processor that was visited by W is said to be "behind it" and any processor that still must be visited by W is said to be "in front of it".

Consider any terminating processor. A processor can terminate only when (it is idle and) it receives a termination signal. By inspecting Rule II (viz.instruction 3a) it is clear that the termination signal can only be generated by some processor p where the latest wave W ended with all token messages white and p idle-white. Consider the moment that p generates the termination signal. Suppose there is a processor q≠p that is still active. Go back in time and consider the moment when W passed q (i.e., when q sent its corresponding token message according to instruction 1). Clearly q must be idle-white at this moment. In order for q to be active later, there must exist a processor q' that is

currently active (and that triggers the chain of activation messages that will eventually activate q). If q' is in front of W, then it can only be idle-black by the time W hits q'. (By inspecting Rule II it is clear that q' an only be idle-white after a wave has passed it.) Thus, when W "passes" q' (or ends there!) q' will color the token message black (or, call for a repeat). This contradicts the fact that W ends white. Hence q' will be behind W. Go further back in time and consider the moment when W passed q'. Clearly q' must be idle-white at this moment. By the same argument one proves that there must be a processor q" behind (the current position of) W. By repeating this it follows that at any moment in time there must have been a processor behind W. This is impossible. Hence no active processor can exist at the moment p generated the termination signal. □

We conclude:

Theorem 3.3. Protocol T is a correct symmetric distributed termination detection algorithm (for networks with a distinguished spanning tree sub-topology and with the link assumption in effect).

We shall now modify protocol T to handle the general case that token messages can bypass each other in a link as well. As explained before, the general case leads to the problem that a wave need not end in one single processor p that can subsequently act as a leader. But it was also explained that the worst that can happen is that a wave "ends" in two neighboring processors p and q, which was referred to as the "double leader" case. A processor that executes instruction 3a of Rule II can determine that it is a double leader in the following way. If a processor p has sent out a token message according to instruction 1 of Rule II, then it cannot receive another token message unless it has received or sent a *-message. So if a token message does arrive from a neighbor q after p sent its token message there without an intermittent *-message, then p knows that it is a double leader and that q is its partner. Thus processors can detect the double leader case at the time they execute instruction 3a, provided they keep track of the token

message that they sent during the current wave. In executing the modified version of instruction 3a in the double leader case, the color of this token message is incorporated in the "$\wedge$" of colors also in deciding termination. If this is done, p and q can act as if they are one leader by virtue of the following observation.

Lemma 3.4. If a wave ends in double leaders p and q, then p and q compute the same color for the wave.

Thus, despite the "distributed" leadership, the double leaders make the same decision about repetition or termination. Clearly the correctness of the termination detection protocol is preserved. We formulate the modified algorithm as "protocol T'".

Protocol T'

{The same assumptions as for protocol T are in effect, except that we now allow token messages to bypass each other in a link.}

Active state:

Rule I'

Exactly as for protocol T.

Idle-X state, X ε {black,white}:

Rule II-X'

Exactly the same instructions as in Rule II-X of protocol T are in effect, except that the following instructions are modified or added in the natural sequence ordering.

2b. After a token message is sent as specified under 1, p keeps track of the neighbor to which it was sent and of the color of the token message (and purges any old information of this sort that it may have).

3a. Modified so as to be applicable only when p is not a double leader.

3b. When token messages $\langle c_1\rangle,\ldots,\langle c_{1+1}\rangle$ (1+1=deg(p))have been received from all different neighbors but p sent out a token message <d> to neighbor q (before a *-message came in or was generated), delete them from the corresponding buffer

queues and do the following. If $c_1 \wedge \ldots \wedge c_{l+1} \wedge d \wedge X$ = white then send a *-message to every neighbor $*_q$ in the spanning tree and go to idle-white.

Clearly instruction 3b handles the double leader case.

Theorem 3.5. Protocol T' is a correct symmetric distributed termination detection algorithm (for networks with a distinguished spanning tree sub-topology).

One wave of protocol T' clearly takes O(N) token messages to run to completion (cf.Hirschberg [19]). Exactly N-1 *-messages are needed to trigger the leaves of the tree to initiate another wave when there is no termination yet. As in section 2, the symmetric DTDA has the property that it only progresses at idle nodes and (thus) expends no unnecessary effort. Finally note that protocol T' is an interesting example of a distributed termination detection protocol for a completely practical model in which there can be no distributed election algorithm. Thus, distributed termination detection is a weaker property than distributed election.

4. A symmetric distributed termination detection algorithm for general networks. We now turn to the design and proof of a symmetric DTDA for general networks. The computational model is very similar as before, except that we no longer assume that the network contains a particular designated sub-topology. Thus, token messages of the protocol can be sent and received over every link of the network, just like activation messages. The general case is interesting because it requires no particular set-up of the network prior to execution, and thus the network can be reconfigured at will between applications without any need of modifying the protocol.

Clearly the general case poses severe problems. There may be many cycles in the network, and we want to "stop" termination detection probes in an efficient manner even though there are no special nodes in

the network (in general). In order that the symmetry can be broken, we assume again that all processors are distinguished by unique but otherwise arbitrary id's. The idea is again to superimpose a termination detection regime on a suitable distributed election algorithm for general networks. We base ourselves on the election algorithm due to Hirschberg[19], because it is natural and quite fast. (It is faster than e.g. the algorithm of Gallager et.al.[17], in the sense that an election completes in "time" proportional to the diameter of the network.) Actually Hirschberg's algorithm as it stands is not very efficient in terms of message complexity: it can take rather long before the ELECT messages of a "non-leader" q are stopped in the network. Thus we use an optimized version in which every processor keeps track of the largest id it has seen in a token message by storing it in an auxiliary register CMAX. Over time the value stored in CMAX will "converge" to the identity of the processor with the largest id in the network (the "leader"), very much as in section 2. No processor will further broadcast incoming token messages with an id smaller than the current value in CMAX. In this way "larger processors" eliminate "smaller processors" much earlier as their id's spread in the network, and we can expect a much smaller message-complexity for the modified protocol. We will return to this aspect later.

We will now give a description of the complete, symmetric DTDA for the general case as "protocol G" below. Recall that token messages have the form <q,c> where q is the identity of the originator (also called the "token id") and c is the color of the token (black or white). Idle processors can be idle-black or idle-white. Initially processors are either active or idle-black.

Protocol G

{The actions are specified for an arbitrary processor p. We will identify p and its id. Token messages may be sent over every link of the network, i.e., no special sub-topology is distinguished for control purposes. The largest token id that was seen at some moment, is maintained in a register CMAX. A processor keeps track of the neighbor(s) from which it received a token message <CMAX,c>

(some c) and to which it sent one, respectively.}

Active state:

{A processor in this state is actively computing on a (sub)task of the distributed application. It can send activation messages to any neighboring processor.}

Rule I

While computing, store incoming messages (of any sort) in the buffer queue of the link over which they come in. Upon finishing the basic computational task, **goto** idle-black.

Idle-X state, X ε {black,white}:

{A processor in this state is idle. It can be reactivated through activation messages (from active processors) only. X is the color of p. Note that repetition of the code below is not automatic, and must be triggered by the receipt of a *-message. If a processor is interrupted to turn active, it resumes the code when it turns idle-black again. We assume that initially CMAX has been set to p.}

Rule II-X

{p acts according to the instructions below. Instruction o has top priority and pre-empts all other instructions when it can be applied.}

o. Store incoming messages (of any sort) in the corresponding buffer queue. If there is an activation message in any queue, then delete it from the queue, initialize p for the appropriate task, and **goto** active.

1. Check the buffer queues for token messages. If there are none and no token message was ever sent by p before, then send out a token message <p,white> to all neighbors. Otherwise, let $\langle q_1, c_1\rangle,\ldots,\langle q_l,c_l\rangle$ (some l with $1\leq l\leq deg(p)$) be the token messages currently received by p. If $\max_{1\leq i\leq l}\{q_i\}<CMAX$, then delete the messages from their queues and, if no token was ever sent by p before, send out a token message <p,white> to all neighbors as well.

   Otherwise...

2a. If $\max\{q_i\} > CMAX$, delete the messages, set CMAX to $\max_{1\leq i\leq l}$

$\{q_i\}$, and send a token message <CMAX, $c_1 \wedge \ldots \wedge c_1 \wedge$ X> to all neighbors except the first that send a token message with id equal to the current value of CMAX to p.

2b. In the latter case, if all neighbors did send a token message with the current (i.e., updated) value of CMAX, then send out a token message <CMAX, $c_1 \wedge \ldots \wedge c_1 \wedge$ X> (necessarily with l=deg(p)) to all neighbors.

3a. If $\max_{1 \leq i \leq l} \{q_i\}$=CMAX, then do the following. If a token message with id CMAX has not been received from every neighbor and none was sent, then delete the token messages, and send a token message <CMAX, $c_1 \wedge \ldots \wedge c_1 \wedge$ X> to all neighbors except the first from which a token message with this id was received.

3b. Continuing this case, if a token message with id CMAX has been received from every neighbor to which p sent out a token message with id CMAX and CMAX>p, then delete the token messages, and send a token message <CMAX, $c_1 \wedge \ldots \wedge c_1 \wedge$ X> to the neighbor to which no token message with this id was sent before. Otherwise (i.e., when a token message with id CMAX has not come in from all directions but p did instruction 3a with the current CMAX), keep the messages in their buffer and wait for more token messages to come in (or...for another instruction to apply).

4. After a black token message is sent out (and only in this case), p goes to idle-white.

5. When token messages $\langle q, c_1 \rangle, \ldots, \langle q_1, c_1 \rangle$ have come in from all different neighbors and all q_i are equal to the current CMAX and CMAX=p, delete the messages from their queues and do the following. If $c_1 \wedge \ldots \wedge c_1$= white, then send a termination signal to all processors, otherwise broadcast a *-message to every processor, and send out a token message <CMAX,white> to all neighbors.

6. When a *-message is received over some links and no *-messages were received before, then delete the *-messages from their queues, delete all send/receive information about all

token messages that ever came in before, and send out *-messages to all neighbors. When a *-message comes in and *-messages were received before, then delete the *-message from its queue. If *-messages have been received on all links, delete the send/receive information about *-messages.

The protocol is considerably more involved than any of the preceding algorithms. To aid in its understanding, it is helpful to consider the subgraphs T_p defined in the following way, for any processor p that started its activity in the protocol by sending out <p,white> messages to all neighbors (by executing instruction 1). The nodes of T_p are the processors which are eventually reached by a token message <p,c> (for some c) and updated their CMAX to p at some stage. For any $q,q' \in T_p$ there is an edge from q to q' if q' first "learned" of p from q. (We assume that q' reads the buffer queues in a particular order so q is unique and well-defined when q' executes instruction 2a.) It is clear that T_p is a rooted tree, tied to the particular execution history of the algorithm, that shows how far token messages with id equal to p were able to spread.

Lemma 4.1

(i) T_p is a spanning tree of the network if and only if p is the leader node (i.e., the processor with largest id).

(ii) Only the leader node can possibly receive its token messages back over all links and thus execute instruction 5.

(iii) The leader node can never turn idle-white.

Proof.

(i) Clearly the leader node will not belong to any tree T_p except his own. On the other hand, if p is the processor with largest id and it eventually turns idle (which we assume), then T_p is defined and the token messages it sends out will eventually "overrun" all nodes and set their CMAX to p (provided all nodes are idle in the long run, which we assume). Thus T_p is a spanning tree of the network.

(ii) Consider any node p that receives token messages <p,..> back over all its links. By inspecting Rule II (viz.instruction 1) p must

have sent out token messages <p,..> to all neighbors at some earlier moment, and no other token messages <p,..> will exist in the network at that time (nor will any be generated by p until the current situation arises). Consider T_p, and suppose it is not a spanning tree of the network. Let q be any leaf of T_p that has a neighbor q' ε T_p. It means that the very first time a token message <p,..> reached q and subsequently caused q to update its CMAX to p (by executing instruction 2a), the token message <p,..> sent out by q to q' (and to perhaps several other neighbors from which no token message with id equal to p was received) according to instruction 2a must have failed to enlist q' in T_p. By inspecting Rule II this necessarily implies that the CMAX of q' must have some value p' with p'>p at the time of receipt. (It can also happen that the CMAX value of q' is updated to p' at this time, because of the receipt of a <p',..> over another link, but this does not alter the essence of the argument.) As a consequence q can never get a token message <p,..> back from q', and thus q can never fulfill the required conditions for sending another token message with id p (by observing instructions 2b and 3). This in turn implies that no node on the path from p to q in T_p can ever return token messages <p,..> and p can impossibly receive its token message back from this direction (i.e., from the direction of q). Contradiction. Thus T_p must be a spanning tree, and by (i) p necessarily is the leader.

(iii) This follows by inspecting Rule II. By instruction 4 a processor can only turn idle-white after sending a black token. As the leader not cannot ever execute instructions 2a and 2b nor any of the sending operations of instructions 3a and 3b, it can only send white token messages and thus never turn idle-white. □

Lemma 4.1.(i) must be interpreted with care. For example, when p is the leader, its T_p will only develop into a full spanning tree if no active processors permanently block its token messages. Also, after the entire network has been "reset" by p's broadcasting of *-messages, the next wave of token messages sent out by p (if there is any) will define a "new" T_p, i.e., a spanning tree with possibly different edges (as defined implicitly by the flow of token messages with id p).

We now prove the correctness of protocol G. It is clear that the protocol never stands in the way of the basic distributed computation and (thus) does not cause deadlock. Thus we only need to consider the properties of the protocol itself.

Lemma 4.2. Protocol G does not deadlock.

Proof.

We argue that the protocol always progresses, unless active processors block the flow of token messages. (Thus we assume that every processor that is active eventually turns idle again, for the sake of argument.) The moment a processor p turns idle, the protocol prescribes some actions for it. Only in one case p is asked to wait (see instruction 3b), namely in the case p did not get a response back from all neighbors to which it sent a token message <CMAX,..> (with the current CMAX). By inspecting Rule II (viz.instructions 2b and 3b) it is clear that the token messages sent out by p will eventually bounce back or be overrun by "larger" token messages (cf.instructions 2a and 3a). In any case the larger token messages will continue to spread and eventually free p from its waiting, either because p is overrun (as in instruction 2) or because it gets the required responses back (as required for instructions 3b and 5). As soon as the leader node steps in, which happens the first time it turns idle, its token messages will spread through the network and kill all smaller token messages in the various buffer queues. (By lemma 4.1.(i) its token messages will reach all nodes of the network.) The leader node will keep the return tokens in the corresponding buffer queues (cf.instruction 3b) until they have come in from all neighbors, and then decide for termination or another wave (cf.instruction 5). Thus the protocol never deadlocks. □

Let p be the leader node. It is instructive to see how its token messages spread through the network and return. Basically the "wave" proceeds down the tree T_p (which was implicitly defined by the flow of token messages), with varying delays in active processors. Note that the sending of token messages is not constraint to certain links and (thus), whenever a processor sends out more token messages, it does not

just send them to its "sons" in the tree. Following instruction 3a the token messages are sent to all neighbors from which no <p,..> message was yet received. As the wave proceeds down the tree, all non-tree links get a token message <p,..> in their queue (but by instruction 3b the processors will patiently keep the token messages in their queues). Consider any leaf node q of T_p. When it first receives the token message(s) with id p, there are two possibilities: either it received the token messages over all its links, or it didn't. In the former case it will return the token message in all directions (by carefully interpreting instruction 3b). In the latter case, it will send a token message to the remaining neighbors (cf.instruction 3a) but, because it is a leaf, fail to be the first to reach them with such a token message. Clearly q can only send the wave back when it receives token messages <p,..> from these neighbors in return (cf.instruction 3b). When any of the neighbors was reached by the wave before q's token message came in, it will have generated the required response and send it to q already (cf.instruction 3a). The other case, in fact, cannot happen because, if the token message sent by q had been the first to reach this neighbor, q would not have been a leaf of T_p! Thus, q will eventually get the required returns on all links and will thus be able to execute instruction 3b with q. All nodes are waiting for the proper returns on all their links (cf.instruction 3b). By the mechanics of Rule II (viz.instruction 3a) and the definition of T_p, token messages are in or "on their way" on all incoming non-tree edges of every node. Thus the "return" wave depends on the generation of token messages <p,..> on the tree-edges back to the root(p), necessarily beginning at the leaves. By the earlier argument every leaf q will eventually be able to execute instruction 3b and send an appropriately colored token message <p,..> back to its father in T_p. Every father of a leaf in T_p that is not the root will thus eventually find its conditions for instruction 3b satisfied and pass a token message <p,..> back to its father, etcetera. Eventually the return wave reaches p and enables it to execute instruction 5, which leads to termination or another wave. Observe that a wave takes exactly 2E token messages, where E is the number of links in the network. (We have omitted the necessary actions of setting all CMAX

registers to p in the first wave it sends out, but this is adequately covered by instruction 2. Observe that initially many processors q may send out waves like p and actually get returns on some links but, by Lemma 4.1, none of these waves can run to completion and none will ever be tried again.)

Lemma 4.3. If all processors are idle, then protocol G will detect it and cause termination.

Proof.

Let all processors be idle. By lemma 4.1 the leader node p is the only processor that can possibly decide termination and broadcast its conclusion through the network. Clearly p must have sent out token messages <p,white> in all directions the first time it turned idle (hence, idle-black). The wave may be colored black in some or all directions, but it is never killed and eventually returns to p like explained above. Consequently, in the current situation p either generates a new wave of token-messages <p,white> (perhaps after some waiting if a return wave was in progress) or decides termination (if the wave in progress returned all white). In the former case observe that idle-black processors will blacken some token messages as the wave proceeds, but that all idle-black processors will turn idle-white after doing so (cf.instruction 4). If there were idle-black processors, then the wave will return to p with black token messages from one or more directions. In this case the next wave p generates will only find idle-white processors in the network and (thus) return all white. It follows that eventually p must receive token messages <p,white> from all directions either the first time or the second time around, thus enabling it to execute instruction 5 and conclude termination. □

Lemma 4.4. No processor can terminate unless all processors are idle or terminated.

Proof.

A processor can terminate only when (it is idle and) it receives the termination signal. By lemma 4.1 the termination signal can only be generated by the leader node p. Consider the moment that p is idle and

"decides" termination. It can do so only because the most recent wave-front W it sent out returned all white from all directions. Let T_p be the rooted tree defined by W's progression through the network. Suppose there is a processor $q \neq p$ that is still active at the moment p decides termination. A contradiction can now be derived, by proceeding with the argument as in the proof of lemma 3.2. Thus no active processor can exist at the moment p concludes termination. □

We conclude:

Theorem 4.5. Protocol G is a correct symmetric distributed termination detection algorithm for general networks.

Protocol G can be optimized in several ways. For example, no separate wave of *-messages need be generated when the leader wants to initiate another termination detection probe (cf.instructions 5 and 6), as the required resetting of the network is implicit in the receipt of new token messages <CMAX,..> after having performed instruction 3b. Also, processors that find token messages waiting in any of their buffer queues the first time they turn idle-black can be prohibited from ever initiating a termination detection probe on their own, regardless of their identity, without loss of generality. As for the protocol in section 2, the latter policy saves unnecessary messages for the implicit leader-finding procedure.

As every processor can in principle start up a termination detection wave and every wave takes (at most) 2E messages, the worst case message complexity of protocol G is 2NE messages before the leader is (implicitly) established. (This is essentially Hirschberg's result [19].) Because of the way we modified Hirschberg's algorithm, it can be expected that the waves of "small" processors are stopped and killed rather quickly, i.e., within a much smaller number of token messages. In the following result we average over all possible id-assignments for a fixed network.

Theorem 4.6. The expected number of token messages generated by

protocol G before the leader node is implicitly established is bounded by O(ElogN).

Proof.

The bound follows by a proper accounting of the token messages that are generated. First we count 2E messages for the spontaneous initiation of the protocol by every processor (cf.instruction 1). Observe that, subsequently, a node will not do any further sending unless it receives a token message with a higher id than is currently in CMAX or satisfies the criterion for sending a return token message (cf.instruction 3b.) Let $X \equiv X_0 X_1 X_2 \ldots$ be the sequence of id's of the processors, starting at a fixed but otherwise arbitrary processor q and proceeding in a fixed breadth-first ordering of the nodes of the network from q. Because we average over all id-assignments of the nodes, X can be considered as a random permutation over 1..N. For the remainder of the argument we refer to some simple facts from the theory of order statistics (see e.g.[16]). We may assume that, on the average, every token message travels at the same speed. Thus the number of times X_0 updates its CMAX register is equal to the number of upper records in X, which is known to be normally distributed and equal to $H_N \approx 0{,}69 \log N$ on the average by results of Feller [12] and Renyi[25]. (H_N is the N^{th} harmonic number.) It follows that an arbitrary node q executes instruction 2a and, hence, sends a token message to $\leq \deg(q)-1$ neighbors at most H_N times on the average and possibly returns a token message at most as many times. This accounts for a total average of $\sum_q \deg(q).H_N = 2E \cdot H_N$ messages. Another 2E messages can be estimated for the complete wave of token messages of the leader node. It follows that the expected message complexity of the protocol is bounded by $O(E) + O(E \cdot H_N) = O(E \log N)$. □

Finally, protocol G can be easily modified so the nodes permanently store the name of the "first" link over which a wave came in and (thus) can restrict the traffic of token messages to the spanning tree T_p after p has been established to be the leader. This would mean that future termination detection probes can be constrained to the spanning tree and thus run at the cost of O(N) instead of O(E) messages per probe.

5. Conclusion. We have shown that for all processor networks that are normally considered, distributed termination detection algorithms can be designed that do not require prior knowledge of a leader node (a root), the size, or any other global knowledge of the network, nor the determination of any global knowledge prior to the actual termination detection procedure. The resulting, fully symmetric distributed termination detection algorithms are all derived by imposing a termination detection regime on a suitable and efficient distributed election algorithm, thus establishing an important paradigm for the design and anlysis of general and message-efficient termination detection protocols.

6. References.

[1] Apt, K.R., correctness proofs of distributed termination algorithms, Rep.84-51, Lab.Informatique Theorique et Programmation, Université Paris 7, Paris, 1984. (To appear in ACM ToPLaS.)

[2] Apt, K.R., and J.L.Richier, Real time clocks versus virtual clocks, Rep.84-34, Lab.Informatique Theorique et Programmation, Université Paris 7, Paris, 1984.

[3] Bodlaender, H.L., and J.van Leeuwen, New upperbounds for decentralized extrema-finding in a ring of processors, Techn.Rep. RUU-CS-85-15, Dept.of Computer Science, University of Utrecht, Utrecht, 1985. Also in: B.Monien & G.Vidal-Naquet(eds.), STACS86-3rd Annual Symposium on Theoretical Aspects of Computer Science, Springer Lect.Notes in Computer Science 210(1986) 119-129.

[4] Bougé, L., Repeated synchronous snapshots and their implementation in CSP, Rep.84-56, Lab.Informatique Theorique et Programmation, Université Paris 7, Paris, 1984. Also in: W.Brauer(ed.), Automata, Languages and Programming - 12th Colloquium, Springer Lect.Notes in Computer Science 194(1985)63-70.

[5] Chandy, K.M., and L.Lamport, Distributed snapshots: determining global states of distributed systems, ACM Trans. Comp,Syst.3(1985)63-75.

[6] Chang, E., and R.Roberts, An improved algorithm for decentralized

extrema-finding in circular configurations of processors, Comm.ACM 22(1979)281-283.

[7] Cohen, S., and D.Lehmann, Dynamic systems and their distributed termination, Proceedings 1st Annual ACM Sympos.on Principles of Distrib.Computing, Ottawa, 1982, pp.29-33.

[8] Dijkstra, E.W., Termination detection for diffusing computations, EWD-687, Burroughs, Nuenen, 1978.

[9] Dijkstra, E.W., W.H.J.Feijen, and A.J.M.van Gasteren, Derivation of a termination detection algorithm for distributed computations, Inf.Proc.Lett.16(1983)217-219.

[10] Dijkstra,E.W., and C.S.Scholten, Termination detection for diffusing computations, EWD-687a, Burroughs, Nuenen, 1979.

[11] Eriksen, O., and S.Skyum, Symmetric distributed termination, DAIMI PB-189, Computer Science Dept., Aarhus University, Aarhus, 1985. Also in: G.Rozenberg & A.Salomaa(eds.)The book of L, Springer Verlag, 1985,pp. .

[12] Feller,W., De fundamental limit theorems in probability theory, Bull.Amer.Math.Soc.51 (1945) 800-832.

[13] Francez,N.Distributed termination,ACM ToPLaS 2(1980)42-55.

[14] Francez,N., and M.Rodeh, Achieving distributed termination without freezing, Techn.Rep.180,Dept.of Computer Science, the Technion, Haifa, 1980. Also in: IEEE Trans.Softw.Engin.SE-8 (1982)287-292.

[15] Francez,N., M.Rodeh, and M.Sintzoff, Distributed termination with interval assertions, in: J.Diaz & I.Ramos(eds.), Formalization of Programming Concepts - Proceedings Internat.Colloq., Springer Lect.Notes in Computer Science 107(1981) 280-291.

[16] Galambos,J., The asymptotic theory of extreme order statistics, J.Wiley & Sons, New York, 1978.

[17] Gallager, R.G., P.A.Humblet, and P.M.Spira, A distributed algorithm for minimum-weight spanning trees, ACM ToPLaS 5(1983) 66-77.

[18] Gouda, M.G., Distributed state exploration for protocol validation, TR 185, Dept.of Computer Science, the University of Texas at Austin, Austin, Texas, 1981.

[19] Hirschberg, D.S., Election processes in distributed systems, preprint, Dept.of Computer Science, Rice University, Houston, Texas, 1980.

[20] Kutten, S., personal communication, 1985.

[21] Lozinskii, E.L., Yet another distributed termination, Techn.Rep.84-2, Dept.of Compuer Science, the Hebrew University, Jerusalem, 1984.

[22] Misra, J., Detecting termination of distributed computations using markers, Proceedings 2nd Annual ACM Symposium on Principles of Distributed Computing, Quebec, 1983, pp.290-294.

[23] Misra, J., and K.M.Chandy, Termination detection of diffusing computations in Communicating Sequential Processes, ACM ToPLaS 4(1982)37-43.

[24] Rana, S.P., A distributed solution of the distributed termination problem, Inf.Proc.Lett.17(1983)43-46.

[25] Renyi, A., Egy megfigyeléssorozat kiemelkedö elemeiröl (On the extreme elements of observations), MTA III. Oszt.Közl. 12(1962)105-121.

[26] Richier, J.L., Distributed termination in CSP-symmetric solutions with minimal storage, Rep.84-49, Lab.Informatique Theorique et Programmation, Université Paris 7, Paris, 1984. Also in: K.Mehlhorn (ed), STACS85-2nd Annual Symposium on Theoretical Aspects of Computer Science, Springer Lect.Notes in Computer Science 18291985)267-278.

[27] Sintzoff, M., Three problems in the design of distributed programs, unpublished notes, IFIP Working Group 2.3, 1978.

[28] Szymanski, B., Y.Shi, and N.Prywer, Terminating iterative solution of simultaneous equations in distributed message passing systems, Proceedings 4th Annual ACM Symposium on Principles of Distributed Computing, Minaki, 1985, pp.287-292.

[29] Tanenbaum, A.S., Computer networks, Prentice-Hall Inc., Englewood Cliffs, NJ, 1981.

[30] Topor, R.W., Termination detection for distributed computations, Inf.Proc.Lett.18(1984)33-36.