Distributed Control Algorithms
(Selected Topics)

Friedemann Mattern

A04/91

Distributed Control Algorithms (Selected Topics)

Friedemann Mattern
FB Informatik, Universität des Saarlandes,
Im Stadtwald 36, D 6600 Saarbrücken, Fed. Rep. Germany
mattern@cs.uni-sb.de

Abstract: The paper presents several algorithmic solutions to typical problems from the theory of distributed computing. The following topics are treated: Distributed approximation, leader election, routing tables for shortest paths, termination detection, parallel graph traversal, information dissemination, consistent snapshot computation. Concepts like atomic actions, message driven computations, time diagrams, and consistent cuts are introduced and references to the literature for further reading are given.

Keywords: Distributed System, Distributed Algorithm, Distributed Termination Detection, Distributed Approximation, Leader Election, Echo Algorithm, C⌃nsistent Snapshot

1 Introduction

The purpose of this paper is to give an overview of some selected topics from the theory of distributed algorithms[1]. *Distributed algorithms* are algorithms which are designed to run on a distributed system where many processes cooperate by solving parts of a given problem in parallel. For this purpose, the processes have to exchange data and synchronize their actions. In contrast to so-called *parallel algorithms*, communication and synchronization is solely done by message passing—there are no shared variables and usually the processes do not even have access to a common clock. Since message transmission times cannot be ignored, no process has immediate access to the global state. Hence, control decisions must be made on a partial and often outdated view of the global state which is assembled from information gathered gradually from other processes.

Concurrent execution, non-determinism (typically introduced by varying message transmission times), and the inability to access the current global state pose some interesting and non-trivial problems. Obviously, classical control problems of parallel shared-memory machines or timeshared pseudo-parallel systems such as deadlock detection or mutual exclusion are more difficult to solve in a truly distributed environment without a central coordinator. However, there are also completely new problems that occur in distributed systems which simply do not exist in sequential or pseudo-parallel systems. A typical example is the so-called snapshot problem. It consists in computing a globally consistent view of the global state of a distributed system without stopping the system. Interestingly, this problem is closely related to a seemingly much simpler problem, namely determining whether a distributed computation (i.e., the execution of a distributed algorithm) has terminated. In fact, determining whether a computation has terminated is

[1]This paper is based on previous work of the author mentioned in the reference list.

a "non-problem" in the sequential world—it is surprisingly difficult, however, to check in a distributed environment whether all processes are passive if a temporarily passive process can be reactivated by the receipt of a message sent by a still active process.

This paper concentrates on *distributed control problems* which arise from the nature of a distributed system itself; we do not discuss application specific algorithms. We introduce the so-called *leader election* problem and the above-mentioned *snapshot* and *termination detection* problems and present several algorithmic solutions. We also explain how it is possible to distributed some data from one process to all other processes on a network of unknown topology. The underlying algorithm, the so-called *echo algorithm*, is also capable to dynamically determine a *spanning tree* and to collect pieces of data distributed among other processes.

1.1 An Example—Solving Cryptarithmetic Puzzles

We start our discussion of distributed control schemes with an algorithm that solves cryptarithmetic word puzzles by means of a parallel *constraint propagation* principle. It is based on an idea of Kornfeld [11], the interested reader may find more about this principle in [16, 18]. The problem consists of mapping the letters of three given strings onto the ten digits such that a correct addition results, e.g.:

```
 LONGER  207563        EUROPE  290782        DONALD
+LARGER +283563       +EUREKA +290234       +GERALD
------- -------       ------- -------       -------
=MIDDLE  491126       =SPIRIT  581016       =ROBERT[2]
```

To enable a parallel solution, each column is represented by a distinct process as depicted in Figure 1. Initially, all letters in all column processes are assigned the maximal set of digits $\{0, ..., 9\}$, and the carry-in and carry-out variables of a process are initialized to $\{0, 1\}$. A column process can receive messages from any other column (or from an external "hypothesis generator") informing it of new constraints on the letters or the carry values. Whenever a column receives a message containing new information, it uses this to find out more about its letters, that is, it tries to compute new constraints. One initial "spontaneous" computation step is performed when the process is established. New constraints on carry-in and carry-out values are sent to the right and left neighbor column respectively and new constraints on letters are sent to all columns which are interested in the information; a simple solution is to broadcast any new information to all processes and to let the receivers decide upon its usefulness.

Notice that many constraint messages can be in transit simultaneously and that the local computations of several column processes can be performed in parallel. This does not mean, however, that the parallel solution is necessarily more efficient than the traditional sequential depth-first search with backtracking—generally a huge number of redundant computations are performed in the parallel version and the message overhead is non-negligible. The constraint propagation method can sometimes narrow the size of the search space drastically, but it may not be able to find a unique solution or a contradiction. If there is more than one possible solution it will never

[2] Left to the reader as a simple exercise. (Hint: D=5)

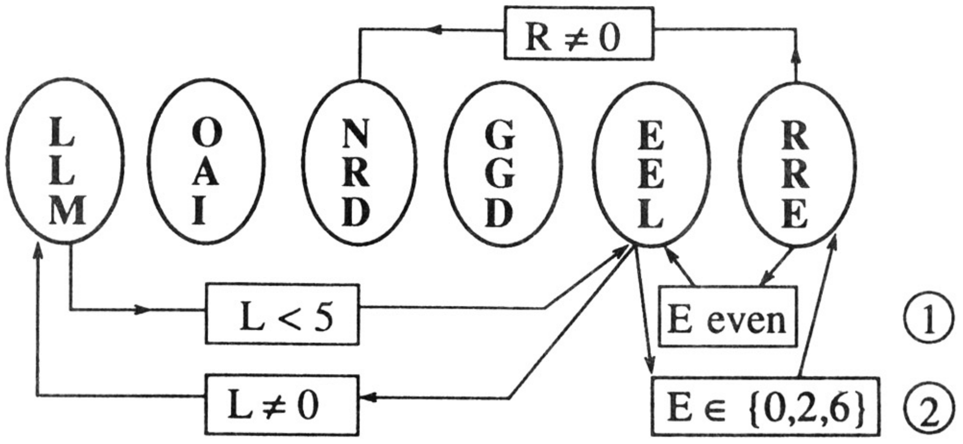

Figure 1: The structure of the distributed puzzle solving system.

find any of them and the resulting sets of digits assigned to the letters will be supersets of all possible solutions. The resulting sets of the example of Figure 1 are $L = \{1,2,3,4\}, M = \{2,3,4,5,6,7,8,9\}, R = \{1,3,5,6,8\}$, and $E = \{0,2,6\}$; all other sets remain unchanged.

Since in general the parallel constraint propagation scheme stagnates without finding a complete solution, a backtracking scheme should be superimposed. The system then works in a sequence of two different phases: The parallel constraint propagation phase is used to prune the search space, and when all constraint activity has terminated a short backtracking phase is started. In this phase either a backtrack step is initiated (when a contradiction has been discovered) or a hypothesis is generated according to some heuristics. A hypothesis acts like an ordinary constraint message which is sent to all relevant columns. This message initiates a new parallel constraint propagation phase.

The crucial problem now is: When does a column process or an external "hypothesis generator" know that the parallel constraint propagation phase has stagnated? Obviously, this is only the case if all processes are waiting for a next message and no more messages are in transit. Since column processes do only react to incoming messages but do not become active or send messages spontaneously, this stagnation state is stable (until some action from outside the system is taken). We call such a state a *termination state.*

As a matter of fact, no process can know a priori whether the distributed computation has terminated. It should also be clear that a method where a process sends control messages to other processes in order to learn whether they are passive (i.e., waiting for a message) is not safe—firstly, possible in-transit messages must also be captured, and secondly, the control messages are probably not all received by the other processes at the same time. Hence, a process which replies that it is passive may be reactivated shortly afterwards by another process that was still active before it was asked about its status.

The parallel puzzle solving algorithm is quite illustrative. It does not only introduce the so-called *distributed termination detection problem* to which we will come back further down, but it is also an example of a message-driven computation which approximates a global result in a distributed way. Such distributed approximation schemes will be discussed in Section 2.

1.2 Actions and Message Driven Algorithms

The execution of a distributed algorithm consists of a parallel execution of several local algorithms which determine the behavior of the processes. Often, the behavior of a process can be understood by specifying its reaction to an incoming message. This reaction consists in the possible sending of messages and an update of the local state. The design and verification of distributed algorithms is simplified if one decomposes the local algorithms into elementary *actions* which transform the state of the process. Actions are considered to be *atomic*, that is, at any instant in time there is at most one action within a process that is executed, and this action cannot be interrupted. Usually, it can also be assumed that all executed actions of a distributed algorithm are globally totally ordered. We discern two types of actions: *internal actions* and *message driven actions*. An internal action **I** of a process p is specified according to the following scheme:

```
I_p:  { guard }
      sequence of statements which do only use the local variables;
      send a message to ...;
      sequence of statements which do only use the local variables;
      ...
```

An action can only be executed if its guard evaluates to true. No message can be received in an internal action, this can only be done at the beginning of message driven actions. Message driven actions are specified similarly:

```
M_p:  { guard }
      receive message ⟨M⟩;
      ...
```

Typically, a process consists of many actions. If several of them are eligible at a given time, a non-deterministic choice is usually assumed. If only message driven actions occur, the distributed algorithm itself is called a *message driven algorithm.*

1.3 The Atomic Model and Time Diagrams

In our model of distributed computations, we do not make any assumptions about the transmission delays of messages. In the sequel, however, we assume that eventually all messages sent are received and all actions terminate. We call a process *active* if it is executing an action, and *passive* else.

A convenient method of visualizing a distributed computation from an abstract and global point of view are so-called *time diagrams*[3]. Figure 2 shows an example: For each process, a horizontal line is drawn in parallel to an imaginary global time axis. Messages are drawn as arrows, and the active phases of a process are highlighted. Since actions are atomic and message delays are arbitrary, we can draw an equivalent time diagram for a given message-driven computation where the duration of the active phases tend to

[3]*"It is true that there are certain implicit dangers in using such graphical representations, because in every geometrical diagram time appears to be misleadingly spatialized. On the other hand, such diagrams, provided we do not forget their symbolic nature, have a definite advantage..."* (Milič Čapek in his philosophical critique [36] on Minkowski's space–time concept.)

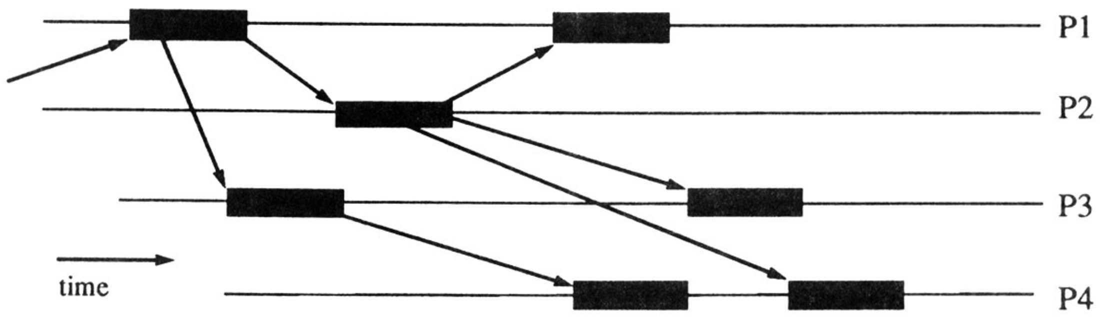

Figure 2: A time diagram.

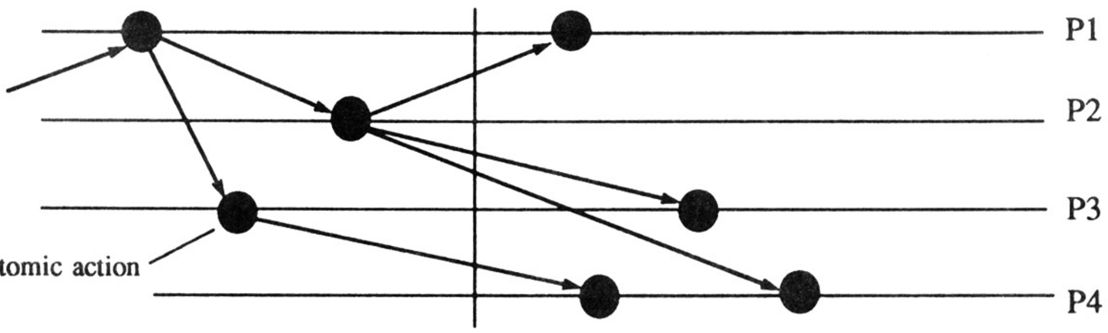

Figure 3: A time diagram for the atomic model.

zero. We do this by assuming that everything happens immediately when an incoming message triggers the execution of an action and by letting the outgoing messages taking somewhat longer on their way to their destination. This yields the *atomic model* of distributed computations. Figure 3 shows a time diagram of an atomic computation which is equivalent to the time diagram shown in Figure 2. The atomic execution of an action is symbolized by a dot.

In the atomic model we get rid of the two process states *active* and *passive*—processes are always "passive". In this model, a process may at any time take any message from one of its incoming communication channels (providing one exists), immediately change its local state, and at the same instant send out any number of messages, possibly none at all. To take a more pictorial view of the atomic model, messages can be thought of flowing steadily but with various speeds towards their destination, eventually hitting a process. Then either the message is quietly absorbed, or new "particles" are ejected, as if from an atomic reaction. Figure 4 illustrates this view.

It should be noted that Figure 4 represents a *snapshot* of a distributed computation at a specific instant in time, whereas a time diagram displays the whole computation in the sense of a sequence of snapshots. In the time diagram depicted in Figure 3, the instant of the snapshot of Figure 4 is marked by a vertical line crossing all process lines. Such *cuts*, which divide a time diagram in two parts, are important for the definition of global states and consistent snapshots (see Section 5). Clearly, in the atomic model a computation is terminated at a given instant t if at that instant no messages are in transit. For a time diagram this signifies that no message arrow crosses the vertical cut line associated to t.

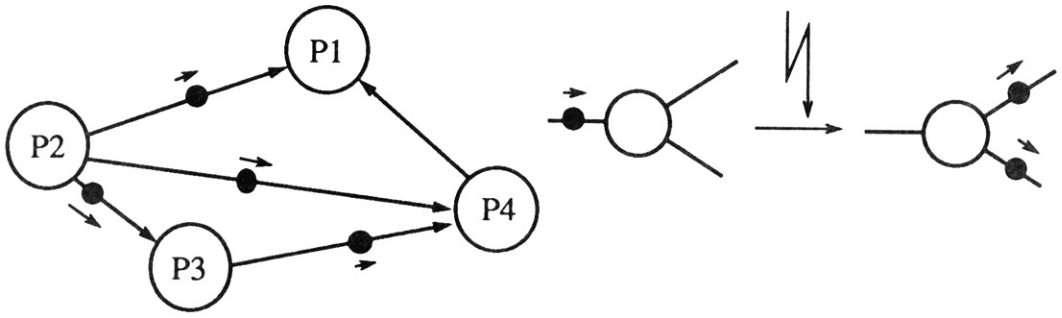

Figure 4: A snapshot of an atomic computation.

2 Distributed Approximation

In the constraint propagation scheme of the puzzle solving algorithm described above, the constraint messages and the local sets of possible digits for the letters of the puzzle can be viewed as *monotonic approximations* of the final result. In fact, a process may only *reduce* a set when considering a newly received constraint, it never augments it. Furthermore, the final values are always included in the local sets or the constraint messages. It turns out that this approximation scheme is a general principle which can be applied to other, more serious problems. The behavior of a process in such a general distributed approximation scheme can be sketched as follows:

```
(0) Compute an initial approximation and distribute it to the other processes.
(1) Wait for a new partial approximation communicated by some other process.
(2) On receipt of such an information
    (a) combine it with the current local approximation and try to compute a better
        approximation,
    (b) if this yields a better approximation then distribute it to the other processes.
(3) Go to step (1).
```

Note that distributed approximation schemes are completely *symmetric* in the sense that all processes behave identically. However, depending on the application the distribution of the new information in steps (0) and (2b) might be restricted to the processes which are "interested" in that information.

In general, the distributed approximation scheme entails a non-deterministic computation because at a given instant in time different messages may be in transit towards the same receiver. Depending on actual message transmission times it may therefore happen that in different executions of the distributed algorithm the messages are received in a different order, thus yielding a different behavior of the algorithm. Interestingly, however, the non-deterministic computations will eventually converge to the same final result in the cases we consider because the combination of the local approximation with the approximation received in a message is an associative, commutative, and idempotent operation.

A general problem with the distributed approximation scheme is its termination. As long as some process has a better approximation than some other process, the computation is not terminated because the second process will eventually receive and process the approximation of the first. However, not every message that is received by a process will entail one or more messages sent by that process—as step (2b) shows, a message might be consumed without any consequence. This guarantees that when "everything has been said" and all processes have the same knowledge, the distributed computation stagnates. Techniques to detect this state will be discussed in later sections. In the following section we will first show how the distributed approximation scheme can be used to elect a leader in an arbitrary network or to compute routing tables for shortest paths.

2.1 Leader Election and Distributed Maximum Finding

The election problem arises whenever several uniquely identified but otherwise "identical" processes have to agree on a *leader*. We assume that process identifications are unique positive integers but that initially no process knows the identities of all other processes. Without loss of generality, the leader to be found is the process (of all processes that participate in the election) with the largest identity. In this case, the election problem is also called *distributed maximum finding*. The election principle is often used as a mechanism for *breaking symmetry* in symmetric distributed algorithms. Other applications are concurrency control or regeneration of a lost unique token. The problem is to design an efficient distributed algorithm which can be initiated by an arbitrary process independently of any other process.

We first consider the problem for arbitrary (strongly connected) networks. This means that every process can reach every other process either directly or indirectly via other processes. We solve the problem using the distributed approximation scheme described in the previous section. The idea is that initially every process considers its own identity p as an approximation of the maximum of all process identities. According to the distributed approximation scheme, this approximation is sent to all neighboring processes. Whenever a process receives a better approximation (i.e., a value higher than it has ever heard of), it again communicates this approximation to all neighbors. This yields the following simple local algorithm for each process (where the local variable M is initialized to 0):

```
$\mathbf{I}_p$:  { $M = 0$ }
      $M := p$;
      send $\langle M \rangle$ to all neighbors

$\mathbf{R}_p$:  { A message $\langle j \rangle$ has arrived }
      if $M < j$ then
         $M := j$;
         send $\langle M \rangle$ to all neighbors;
      fi

$\mathbf{T}_p$:  { Termination has been detected }
      if $M = p$ then "I am the leader" fi
```

Of course, it is not necessary that in action $\mathbf{R}_p$ process p returns the value j it has just received to the sender of that value. The guard $M = 0$ in the initial action $\mathbf{I}_p$ serves two purposes. Firstly, it guarantees that a process starts the algorithm at most once, and secondly, it excludes the possibility that a process initiates the algorithm when a leader has already been determined. Otherwise, however, an arbitrary number of initiators may start the algorithm concurrently, and a process may join an election until it receives the first message. Notice that after termination all local variables M have the same value, namely the identity of the leader. That is, once the computation has terminated, every process knows the identity of the leader, but no process "knows that it knows" this value.

Interestingly, the termination detection problem can easily be solved if the election algorithm is executed on a (unidirectional) *ring* instead of an arbitrary network. In this case every process has exactly one neighbor; the local algorithm for each process is only slightly different from the generalized version:

```
I_p:  { $M = 0$ }
      $M := p$;
      send $\langle M \rangle$ to neighbor

R_p:  { A message $\langle j \rangle$ has arrived }
      if $M < j$ then
         $M := j$;
         send $\langle M \rangle$ to neighbor;
      fi;
      if $j = p$ then "I am the leader" fi
```

The idea of this principle, which was already published in 1979 by Chang and Roberts [3], is simple: A message sent around the ring is eliminated by the first larger process encountered which participates in the election. Thus all messages except the message with the highest value are eliminated on their way around the ring. A process which gets back a message with its own identification has won the election because this message was not eliminated by any other process.

The average message complexity of the unidirectional ring based variant is $nH_k \approx n \log k$, where n denotes the total number of processes, k denotes the number of initiators, and H_k is the k-th harmonic number. In fact, the algorithm is average case optimal [22], its statistical behavior is analyzed in [17]. The worst case message complexity is $nk - k(k-1)/2$ or $n(n+1)/2$ if all processes are initiators, i.e., if $n = k$. For ***bidirectional rings***, where a process initially makes a random choice for the direction to which it sends its message, the average message complexity is even somewhat lower [14, 17]. There exist several other ring-based election algorithms with a lower worst case message complexity, see [23, 35].

The election scheme for arbitrary connected networks described above can easily be generalized to compute *routing tables for shortest paths* in a distributed way. Here we only sketch the idea, the details are left to the reader. Assume that for a given (undirected and connected) graph each edge is assigned a cost value. A node is represented by a process which can send messages to all neighboring nodes. Each process p maintains a *routing table* which consists of entries $(id, cost, via)$, one for each process. Initially, all entries for which id is the identification of a neighboring node are of the form (id, c, id) where c is the cost assigned to the edge from p to id. All other entries are of the form

$(id, \infty, ?)$. Initially, a process sends its own routing table to all neighbors. A process which receives a message with a routing table first adds the cost of the edge connecting it with the sender to all values of the cost field in the received table. It then compares the table to its own table. If it learns about a shorter path to some node via the sender of the message, it updates its routing table and sends it to all other neighbors.

The reader may easily check for a simple graph that the algorithm does indeed compute shortest or "cheapest" paths in a distributed way. Unfortunately, we have again the termination detection problem. For this specific problem one might think of solutions based on acknowledgements such that a process is informed when all direct and indirect activities in other processes for which it is responsible have ceased. While there exist solutions based on this and other principles [29], we are mainly interested in solutions to the distributed termination detection problem which are independent of the application. Such solutions will be presented in the next section.

3 The Termination Detection Problem

We have seen that in general it is non-trivial to decide whether a distributed computation has reached a state where no process is active and no messages are in transit. This is due to the fact that in a distributed system no process has a consistent and up to date view of the global state. Although the termination detection problem is simple to formulate, a surprising variety of algorithms with rather different properties has been published in recent years. For those published before 1987 we refer to the bibliography in [15].

The problem of termination detection is to superimpose on a given so-called *basic computation* a *control computation* which enables one or more of the processes to detect when the termination condition holds for the basic computation. The following two criteria specify the correctness of the control algorithm.

Safety. If the control algorithm detects termination, then the termination condition holds.

Liveness. If the termination condition holds, then eventually the control algorithm will detect it.

For the solution of the problem we confine ourselves to the atomic model introduced in Section 1.3 where processes are never active. This is justified because it is not necessary that the superimposed detection algorithm ever "sees" that a process is active. More precisely, a detection algorithm which "visits" the processes (i.e., where the processes have to react to control messages) may simply be suspended in an active process and reactivated after the process has become passive. Usually, this does not make the termination detection algorithm less efficient because it seems to be plausible that as long as a process is active the algorithm cannot do anything more reasonable than waiting for the process to become passive. Although not all termination detection algorithms are "visit-based"[4], we confine ourselves to such algorithms in this paper.

In the atomic model the termination condition simply reads *"there are no messages in transit"*. An obvious attempt to test this condition is to let every process count

[4]A well-known algorithm which does not rely on this paradigm is for example the diffusing computations scheme of Dijkstra and Scholten [7].

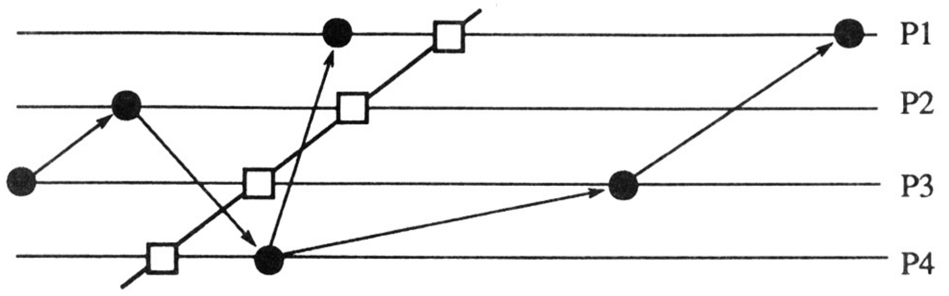

Figure 5: A counter-example to the simple counting scheme.

the number of sent and received basic messages (i.e., messages of the underlying basic computation). For that, every process P_i keeps a counter s_i for the number of sent basic messages, and a counter r_i for the number of received basic messages. The control algorithm could now visit the processes and accumulate the counters by $S := \sum_i s_i$ and $R := \sum_i r_i$. Unfortunately, however, it is wrong to conclude from $S = R$ that the computation has terminated!

Figure 5 shows a counter-example to the naive counting scheme. In this time diagram the visit-actions of the control algorithm are marked by a square and are connected by a line. For the values collected along this line we have $S = 2$ and $R = 2$ although the computation is not terminated after the last process has been visited! As one might notice this is due to a *compensating effect* caused by messages sent to the right of the line and received to the left of that line. Obviously, since messages do not flow backwards in time, such an effect is only possible if the line is not vertical. Unfortunately, it is not possible to enforce that all processes are visited by the control algorithm simultaneously[5]. Hence, we conclude that the simple counting principle does not fulfill the safety property—it is wrong.

3.1 The Four Counter Solution

Surprisingly, the wrong scheme sketched in the previous section can be transformed into a correct termination detection algorithm by applying it twice. The idea is to start a second "round" after completion of the first, yielding the accumulated counter values S' and R'. We claim that the termination condition holds if the values of the four counters acquired in the two rounds are equal, i.e., if $S = R = S' = R'$.

A formal proof of the safety property is given in [15], here we only present an informal argument. The underlying idea is that because of the "gap" between the two cut lines (i.e., the interval between the end of the first round and the start of the second, see Figure 6), it is not possible that a message sent to the right of the second line is received to the left of the first line. Hence, it is not possible to compensate the counters of both rounds at the same time. This argument can be made more precise by observing that when $S = S'$ holds, then no message is sent between the two cut lines. Similarly, $R = R'$ implies that no message is received between the two cut lines. Hence, if this is the case, then throughout the interval between the two cut lines the number of messages sent is

[5]It is possible, however, to simulate such an instantaneous vertical cut by freezing the underlying computation before starting a visit. Since this seriously affects the underlying computation, this is not a recommendable solution, though.

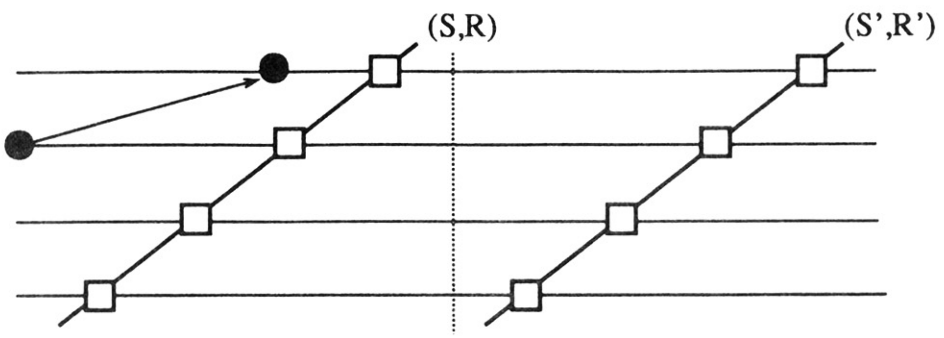

Figure 6: The four counter termination detection principle.

S and the number of messages received is R. Therefore, $S = R = S' = R'$ implies that for any instant *in the gap*, the number of messages sent equals the number of messages received. This means that $S = R$ holds for the dashed *vertical* cut line in Figure 6, hence no messages can be in transit.

The liveness property is easily verified: Once the termination condition holds, all subsequent runs of the counter accumulating scheme will return the same value for the number of messages sent and received. A nice property of this termination detection algorithm is that no process variables are changed in the control actions of the algorithm (which are executed when a process is visited by a control message). Hence, different executions of the algorithm, possibly started by different initiators, do not interfere.

3.2 A Solution for Synchronous Communications[6]

We now consider the distributed termination detection problem for a model which is different from the atomic model. Here we assume that the basic messages are transmitted *instantaneously* and are therefore never in transit. This is an abstract view of computations with *synchronous* communications [5], languages like CSP or Occam are based on such a synchronous model. In this model, the basic computation behaves according to the following rules:

($\mathcal{R}1$) Only an active process is allowed to send messages.

($\mathcal{R}2$) A passive process becomes active when it receives a message.

($\mathcal{R}3$) At any time, a process may change from active to passive.

It is usually assumed that initially at least one process is active. Clearly, when the basic computation has reached a state where all processes are passive, no basic messages will be sent any more, and all processes will remain passive forever. Hence the termination condition for this model is *"all processes are passive"*. Interestingly, this model can easily be transformed to the atomic model by assuming that when a process becomes active, a virtual message is sent which is only received after the process is passive again. The reader may therefore easily adapt the four counter solution presented in the preceding section to the synchronous model. Here, however, we aim at a different solution which is not based on counting.

[6]This section is based on the paper "Global Virtual Time Approximation with Distributed Termination Detection Algorithms" [21], co-authored by H. Mehl, A. Schoone, and G. Tel.

Formally, we model the behavior of the basic computation by means of three atomic actions such that rules $\mathcal{R}1$, $\mathcal{R}2$, and $\mathcal{R}3$ are obeyed. We assume that each process P_i has a system variable $state_i$ with values from $\{active, passive\}$. Rule $\mathcal{R}3$ corresponds to the internal action $\mathbf{I}_i$:

```
I_i:  $state_i := passive$
```

The transmission of a message to a process P_j is described in action $\mathbf{X}_i$ where we have to take rule $\mathcal{R}1$ into account:

```
X_i:  if $state_i = active$ then send $\langle\cdots\rangle$ to $P_j$ fi
```

The receipt action $\mathbf{R}_i$ of the basic computation reflects that a process becomes activated by the receipt of an activation message (rule $\mathcal{R}2$):

```
R_i:  receive $\langle\cdots\rangle$;
      $state_i := active$
```

The basic computation is terminated if $\forall i$, $state_i = passive$. For a first attempt to solve the termination detection problem in this model, assume that each process P_i has a state indicator S_i always correctly reflecting the state of the process (i.e., $S_i = state_i$). Then an initiator may start a *control wave* which visits all processes and returns the values of the state indicators. In the simplest case, a control wave is implemented by a control message which travels on a (virtual) ring connecting all processes. A more sophisticated possibility would be to use the echo algorithm presented in Section 4.

Unfortunately, however, the values of the state indicators collected in that way are not useful for termination detection. Because of possible reactivations of processes "behind the back" of the wave, the observation that all processes were passive when being inspected by the wave does not imply that all processes were passive simultaneously. (Notice that even for instantaneous control messages the complete execution of the control wave algorithm is not instantaneous because the wave may be delayed at processes it inspects. Therefore, processes may be reactivated while the wave is in progress.)

Fortunately, the simple scheme can easily be transformed into a correct algorithm. Assume now that the state indicators S_i are "sticky" in the following sense. If a process P_i is activated, the value of S_i becomes (or remains) active. If a process becomes passive, however, S_i "sticks" to active. Before the start of the termination detection algorithm, the state indicators of all processes should be initialized to the value of $state_i$, thus correctly reflecting the state. To implement the sticky state indicators we only need to augment the receipt action $\mathbf{R}_i$ of the basic computation with the proper assignment to S_i:

```
R_i:  receive $\langle\cdots\rangle$;
      $state_i := active$; $S_i := active$
```

As S_i is *not* set to *passive* when the state of the process becomes *passive*, the internal action $\mathbf{I}_i$ is not changed, nor is action $\mathbf{X}_i$.

Clearly, if at the start of the control wave some process P_j was active, the algorithm will not announce global termination because the value of S_j is still *active* when it is eventually collected by the wave. Or, to put it in another way: If the algorithm reports termination, then no process was active at the start of the wave; hence the basic computation has actually terminated because it was already terminated when the wave was started. This shows that the implicit semantics of the sticky state indicators ensures the *safety* of the resulting termination detection algorithm.

Unfortunately, however, in the scheme as it stands termination will never be announced unless all processes were initially passive. To guarantee *liveness* it is necessary to repeatedly first reset the sticky state indicators to the true values of their processes' states and then start a new control wave. Then, when the basic computation terminates, eventually the sticky state indicators will be set to *passive* (and never reset to *active*). Consequently, termination will be announced at the end of the next wave. However, in order not to compromise the safety property, a state indicator must not be reset to *passive* between the start of a wave and the collection of its value.

The "sticky-flag" scheme can easily be realized using a circulating control message. In that case a dedicated process, P_n, initiates the algorithm by sending a control message to the next process (i.e., P_1) on the ring:

```
send $\langle passive \rangle$ to $P_1$
```

The circulating control message is only accepted by a process P_i when the process is passive. It then executes the following atomic action:

```
W_i:     { $state_i = passive$ }
   (1)   receive $\langle M \rangle$;
   (2)   if $S_i = active$ then $M := active$ fi;
   (3)   if $i = n$ and $M = passive$ then signal termination
   (4)                                 else if $i \neq n$ then send $\langle M \rangle$ to $P_{i+1}$
   (5)                                                else send $\langle passive \rangle$ to $P_1$
   (6)                                      fi;
   (7)                                      $S_i := passive$
   (8)   fi
```

In line (1) the contents of the received message is assigned to M. In line (2) M "accumulates" the value of the state indicator S_i. If after a complete round M is still *passive* (3), termination can be signaled. Otherwise, the control message is propagated (4); at P_n, however, it must be reinitialized to *passive* (5). In line (7) the state indicator is reset to *passive* in order to guarantee liveness as discussed above.

The algorithm is reminiscent of the well-known termination detection algorithm by Dijkstra, Feijen, and van Gasteren [6]. However, whereas in that algorithm a flag is set when a message is *sent*, our scheme uses a flag (the sticky state indicator) which is set when a message is *received*. More details and a formal proof of a generalized variant of the "sticky flag" algorithm may be found in [21].

4 The Echo Algorithm

The echo algorithm is a distributed algorithm which can be used to traverse an arbitrarily connected graph in a parallel way and to distribute data of the initiator to all processes. It was first published by Chang in 1982 [4]; Segal found a slightly more efficient version in 1983 [30]. The basic idea is to "flood" the graph by sending messages (so-called *explorers*) to all neighbors and propagating them further on, but also to keep track of the sender of a message in order to be able to send an acknowledgement. Acknowledgements are called *echoes*, they can be used to carry data from the processes back to the initiator of the algorithm. Implicitly, the edges traveled by the echoes build a *spanning tree* of the graph.

The algorithm can be sketched as follows. An *initiator* sends out explorers to all its neighboring processes. Upon receipt of the first explorer a process becomes *engaged* and propagates the "explorer wave" further on to all its neighbors. A process that has no other communication link than the one along it has received the explorer immediately returns an echo. Having received explorers or echoes along every incident communication link, a process becomes disengaged and returns an echo to the process from which it was engaged. Eventually the "echo wave" reaches the initiator. More formally, the local algorithm of each process P_i can be specified as follows with two atomic actions:

```
X_i:  receive ⟨ECHO⟩ or ⟨EXPLORER⟩ from p;
      if ¬ ENGAGED then
              ENGAGED := true;          /* become red */
              N := 0; PRED := p;
              send ⟨EXPLORER⟩ to NEIGHBORS\{PRED}
      fi;
      N := N+1;
      if N = |NEIGHBORS| then
              ENGAGED := false;         /* become green */
              if initiator then terminated
                        else send ⟨ECHO⟩ to PRED
              fi
      fi

I_i:  { ¬ ENGAGED }
      initiator := true;
      ENGAGED := true;
      N := 0;
      send ⟨EXPLORER⟩ to NEIGHBORS;
```

Here, NEIGHBORS denotes the set of neighbors, and |NEIGHBORS| its cardinality. Variable N is a counter for the explorers and echoes, and PRED stores the identity of the process to which an echo must eventually be sent back. The boolean variables ENGAGED and *initiator* should be initialized to **true** in every process. Action $\mathbf{I}_i$ is used to start the algorithm; at most one process should start it. If more than one process should have the possibility to start the algorithm then the processes have to synchronize using an election scheme or a distributed mutual exclusion algorithm.

The echo algorithm can be used to implement waves of control messages which visit all processes. Such *control waves* are needed for many control algorithms, examples were given in Section 3. To "see" the waves, it is instructive to describe the algorithm using three colors *white*, *red*, and *green*. Assume that originally all processes and edges of the graph are white. The initiator turns red upon the start of the algorithm. Explorers are red, echoes green. A message colors edges (along which it travels) in its own color. A red message arriving at a white process colors that process red. A process having received red or green messages along all its incident edges becomes green (before it sends an echo). Clearly, every process changes from white via red to green, for processes with a single edge the red phase is rather short. It is easy to see that after the execution of the algorithm the green edges (which were first colored red by a message moving in one direction and later green by an opposite echo message) build a spanning tree of the graph. Edges remaining red were colored simultaneously by two red messages moving in opposite directions. Since every edge is traveled by exactly two messages (an explorer in one direction, and an echo *or* an explorer in the other direction) the message complexity of the algorithm is $2e$ if e denotes the number of edges.

Obviously, a single run of the echo algorithm realizes two waves, an exploding red "explorer wave" and a contracting green "echo wave". A process is visited by the first wave when it changes its color from white to red, and it is visited by the second wave when it changes its color from red to green. Interestingly, the two waves are not completely separated—it is possible that some parts of the graph are still engaged with the red wave while in other parts the green wave is already in progress. It is always the case, however, that a green process does not have a white neighbor. This feature can for example be used in the four counter termination detection algorithm where a single run of the echo algorithm can implement the two necessary "rounds" of the termination detection scheme. This is possible because the echo algorithm guarantees that a basic message sent after the second round (implemented by the green wave) cannot be received before the first round (implemented by the red wave). Other algorithms which implement waves may be found in [27].

The echo algorithm is only one instance of a class of algorithms referred to as *total algorithms* [31, 32]. As their name indicates, total algorithms involve all processes of a distributed system, most of these algorithms can be used to implement waves on which another control algorithm is based. Thus, algorithms for the solution of control problems can often be designed in a modular way: An abstract problem specific part (e.g., counting twice the number of sent and received messages to detect termination), and another part which implements the waves.

5 The Consistent Snapshot Problem

In this section we briefly discuss the so-called *snapshot problem.* The problem is to determine a "meaningful" view of the global state of a distributed computation. Because in a distributed system no process has immediate access to the global state or to the local state of another process, this is a non-trivial problem. Fortunately, there exist algorithms which determine such a view for a suitable interpretation of "meaningful" without stopping the underlying system.

Snapshots and snapshot algorithms are fundamental paradigms of the theory of distributed computing. Important applications are algorithms for the detection of *stable properties* (i.e., properties which remain true once they become true) such as deadlocks of distributed systems [2]. Since in the case of the atomic model "no messages are in transit" and in the case of the synchronous model "all processes are passive" are stable properties of the global state, snapshot algorithms can also be used to solve the termination detection problem. More generally, snapshot algorithms can be used to compute lower bounds of monotonic functions of the global state such as the simulation time to which a distributed simulation system has advanced (the so-called *Global Virtual Time*) [8, 9, 21]. Other applications are checkpointing and recovery of distributed data bases and monitoring and debugging of distributed systems.

A snapshot of the global state consists of the local states of all processes and the messages in transit. It is usually required that a snapshot is meaningful in the sense that it corresponds to a *possible* global state which could have occurred in the computation if the local states of all processes and all communication channels were recorded simultaneously. In order to get such a causally consistent state in a system without a common clock the local state recording actions must be coordinated: If the receipt of a message is recorded the corresponding sending of the message (which usually takes place at another process) must also be recorded. More generally, all "causal predecessors" of a recorded action must also be recorded. Fortunately, this is indeed possible without freezing the whole system.

A first snapshot algorithm was presented by Chandy and Lamport for systems with FIFO channels [1]. The main idea is that immediately after recording the local state a process sends control messages along each of its channels. Whenever a process receives a control message for the first time it takes a local snapshot (i.e., it records its state). Causal consistency is guaranteed due to the FIFO property of the channels because any message of the underlying application sent after the control messages must arrive after the local snapshot of the receiver. Messages in transit can easily be recorded because control messages flush the channels.

The local state recording events of a snapshot algorithm define a *cut* which can be represented as a line cutting a time diagram in two parts. Informally, a cut is *inconsistent* if there exists a message sent to the right of that line which is received to the left of it, otherwise a cut is *consistent*. Snapshots taken along an inconsistent cut line are not meaningful—they would show the receipt of a message but not its sending[7]. Therefore, the determination of a consistent cut is central to any snapshot algorithm. In [12] Lai and Yang present a simple scheme to compute a consistent cut for non-FIFO systems by piggybacking a one bit status information (encoding the two colors white and red) onto basic messages:

1. Every process is initially white and turns red while taking a local snapshot.
2. Every message sent by a white (red) process is colored white (red).
3. Every process takes a local snapshot at its convenience–but before a red message is possibly received.

These rules can be implemented by a snapshot algorithm as follows. Assume that there is a single initiator only. The initiator becomes red and then starts a virtual broadcast algorithm by directly or indirectly sending (red) control messages to all processes in order to ensure that eventually all processes become red. Virtual broadcast algorithms can be implemented in various ways, for example by using control messages propagated along

[7]Observe that the cut line in Figure 5 illustrating the counter-example to the simple counting termination detection scheme is inconsistent. In fact, simple counting is correct if the counters are accumulated along a *consistent* cut line.

rings or spanning trees, or by using flooding schemes such as the echo algorithm. A white process takes a local snapshot when it receives such a control message. Of course, it may happen that a white process receives a red basic message before receiving a control message. A process must always be prepared to this case, and if this happens it must take a local snapshot at the moment it receives a red basic message (before executing the action triggered by the message).

For a complete snapshot the messages in transit must also be taken into consideration. To catch those messages, Lai and Yang propose that a process keeps a record of all messages sent and all messages received along its channels. These message histories are part of the local states. After the local snapshots have been "assembled" the messages in transit can be determined for each channel by computing the difference of sent and received messages. A serious drawback of this method, however, is that complete message histories must be stored and communicated which might require a large amount of space.

There is a different method to catch the messages in transit which does not suffer from this drawback [20]. Obviously, the messages in transit are precisely the white messages which are received by red processes. Therefore, whenever a red process gets a white message it can send a *copy* of it to the snapshot initiator. After the snapshot initiator has received the last copy (and the local snapshots of all processes) it knows the complete snapshot. A problem with the method described so far, however, is termination detection. The initiator successively gets copies of all messages in transit but it does not know when it has received the last one.

Fortunately, this problem can easily be solved by applying a distributed termination detection algorithm to the white messages only (including the copy messages). When no more white messages are in transit (and no process is white any more) the snapshot initiator knows that it has received the last copy message and that its snapshot is complete and consistent. Depending on the termination detection algorithm used, different snapshot algorithms result. This idea is discussed in more detail in [20].

6 Further Reading

In this paper we have only sketched some topics, many other interesting aspects of the field of distributed control algorithms could not be discussed. For example, we did not show how such fundamental notions like consistency, causality, logical time, and synchrony are related to the theory of distributed algorithms [5, 13, 19], and we did not treat the aspect of verification of distributed algorithms [29]. We did also not treat the *distributed deadlock detection problem* or the *distributed mutual exclusion problem.* Here, the reader is referred to [10] and [28] where several algorithmic solutions are discussed and further references are given. Another interesting class of algorithms are *distributed garbage collection* schemes, in [34] several solutions are presented, further references are given, and it is shown that every garbage collection algorithm can be transformed mechanically into a distributed termination detection algorithm.

Up to date information on the theory of distributed algorithms may be found in the yearly proceedings of the International Workshop on Distributed Algorithms which are usually published by Springer-Verlag in the LNCS series. Until now, there exist only few monographs on distributed algorithms [18, 24, 25, 32], but a few more promising books treating the subject as a whole are expected to appear in 1992 [26, 33].

References

[1] K.M. CHANDY and L. LAMPORT. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems 3:1, pp. 63-75*, 1985.

[2] K.M. CHANDY, J. MISRA, and L.M. HAAS. Distributed Deadlock Detection. *ACM Transactions on Computer Systems 1:2, pp. 144-156*, 1983.

[3] E. CHANG and R. ROBERTS. An Improved Algorithm for Decentralized Extrema-Finding in Circular Configurations of Processes. *Comm. of the ACM 22:5, pp. 281-283*, 1979.

[4] E.J.H. CHANG. Echo Algorithms: Depth Parallel Operations on General Graphs. *IEEE Transactions on Software Engineering SE-8:4, pp. 391-401*, 1982.

[5] B. CHARRON-BOST, F. MATTERN, and G. TEL. Synchronous and Asynchronous Communication in Distributed Systems. *Technical Report, Université Paris 7, Paris*, 1991.

[6] E.W. DIJKSTRA, W.H.J. FEIJEN, and A.J.M VAN GASTEREN. Derivation of a Termination Detection Algorithm for Distributed Computations. *Information Processing Letters 16, pp. 217-219*, 1983.

[7] E.W. DIJKSTRA and C.S. SCHOLTEN. Termination Detection for Diffusing Computations. *Information Processing Letters 11:1, pp. 1-4*, 1980.

[8] R.M. FUJIMOTO. Parallel Discrete Event Simulation. *Comm. of the ACM 33:10, pp. 30-53*, 1990.

[9] D.R. JEFFERSON. Virtual Time. *ACM Transactions on Programming Languages and Systems 7:3, pp. 404-425*, 1985.

[10] E. KNAPP. Deadlock Detection in Distributed Databases. *Computing Surveys 19:4, pp. 303-328*, 1987.

[11] W.A. KORNFELD. The Use of Parallelilsm to Implement a Heuristic Search. *In: Proc. of the International Joint Conference on Artificial Intelligence, pp. 575-580*, 1981.

[12] T.H. LAI and T.H. YANG. On Distributed Snapshots. *Information Processing Letters 25, pp. 153-158*, 1987.

[13] L. LAMPORT. Time, Clocks, and the Ordering of Events in a Distributed System. *Comm. of the ACM 21:7, pp. 558-565*, 1978.

[14] C. LAVAULT. Average Number of Messages for Distributed Leader Finding in Rings of Processors. *Information Processing Letters 30, pp. 167-176*, 1989.

[15] F. MATTERN. Algorithms for Distributed Termination Detection. *Distributed Computing 2, pp. 161-175*, 1987.

[16] F. MATTERN. Experience with a New Distributed Termination Detection Algorithm. *In: Van Leeuwen J. (ed) Proc. of the 2nd International Workshop on Distributed Algorithms, Springer-Verlag LNCS 312, pp. 127-143*, 1988.

[17] F. MATTERN. Message Complexity of Simple Ring-based Election Algorithms – an Empirical Analysis. *In: Proc. 9th International Conference on Distributed Computing Systems, pp. 94-100*, 1989.

[18] F. MATTERN. Verteilte Basisalgorithmen. *Springer-Verlag, Informatik-Fachberichte Bd. 226*, 1989.

[19] F. MATTERN. Virtual Time and Global States of Distributed Systems. *In: Cosnard M. et al. (eds): Proc. Workshop on Parallel and Distributed Algorithms, Château de Bonas Oct. 1988, Elsevier, pp. 215-226*, 1989.

[20] F. MATTERN. Efficient Distributed Snapshots and Global Virtual Time Algorithms for Non–FIFO Systems. *Technical Report SFB124-24/90, University of Kaiserslautern*, 1990.

[21] F. MATTERN, H. MEHL, A.A. SCHOONE, and G. TEL. Global Virtual Time Approximation with Distributed Termination Detection Algorithms. *Technical Report RUU–CS–91-32, University of Utrecht*, 1991.

[22] J. PACHL, E. KORACH, and D. ROTEM. Lower Bounds for Distributed Maximum-Finding Algorithms. *Journal of the ACM 31:4, pp. 905-918*, 1984.

[23] G.L. PETERSON. An O(nlogn) Unidirectional Algorithm for the Circular Extrema Problem. *ACM Transactions on Programming Languages and Systems 4:4, pp. 758-762*, 1982.

[24] M. RAYNAL. Algorithmes Distribués & Protocoles. *Editions Eyrolles, Paris (also: "Distributed Algorithms and Protocols", Wiley, 1988)*, 1985.

[25] M. RAYNAL. Systèmes Répartis et Réseaux - Concepts, Outils et Algorithmes. *Editions Eyrolles, Paris (also: "Networks and Distributed Computation", MIT-Press, 1988)*, 1987.

[26] M. RAYNAL. *Synchronisation et Etat Global dans les Systèmes Répartis.* Eyrolles, 1992.

[27] M. RAYNAL and J.-M. HELARY. Control and Synchronisation of Distributed Systems and Programs. *Wiley*, 1990.

[28] B.A. SANDERS. The Information Structure of Distributed Mutual Exclusion Algorithms. *ACM Transactions on Computer Systems 5:3, pp. 284-299*, 1987.

[29] A.A. SCHOONE. Assertional Verification in Distributed Computing. *Dissertation, University of Utrecht*, 1991.

[30] A. SEGAL. Distributed Network Protocols. *IEEE Transactions on Information Theory IT-29:1, pp. 23-35*, 1983.

[31] G. TEL. Total Algorithms. *In: Vogt F.H. (ed) Concurrency 88, Springer-Verlag LNCS 335, pp. 277-291*, 1988.

[32] G. TEL. *Topics in Distributed Algorithms*, volume 1 of *Cambridge International Series on Parallel Computing.* Cambridge University Press, Cambridge, U.K., 1991.

[33] G. TEL. *Introduction to Distributed Algorithms.* To appear, 1992.

[34] G. TEL and F. MATTERN. The Derivation of Distributed Termination Detection Algorithms from Garbage Collection Schemes. *Technical Report RUU-CS-90-24, Dept. of Computer Science, University of Utrecht*, 1990.

[35] J. VAN LEEUWEN and R. TAN. An Improved Upperbound for Distributed Election in Bidirectional Rings of Processors. *Distributed Computing 2, pp. 149-160*, 1987.

[36] M. ČAPEK. Time–Space Rather than Space–Time. *Diogenes*, (123):30–49, 1983.