Total Algorithms

Gerard Tel

RUU–CS–88–16
April 1988

Rijksuniversiteit Utrecht

Vakgroep Informatica

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

Total Algorithms

Gerard Tel

Department of Computer Science, University of Utrecht,
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands.

Abstract: We define the notion of *total algorithms* for networks of processes. It turns out that total algorithms are an important building block in the design of distributed algorithms. For many network control problems it can be shown that an algorithm solving it is necessarily total, and that any total algorithm can solve the problem. We study some total algorithms and compare their merits. Many known results and algorithms are thus put in a general framework.

Key words and phrases: Network control problems, Broadcast, Election, Distributed Infimum, Resynchronization.

1 Introduction

The design and verification of distributed algorithms is a difficult task. To make this task easier, modular design techniques have been advocated recently, see e.g. Gafni [Ga84], Segall [Se83], and Tel [Te86]. Modular design techniques not only facilitate the design and verification of distributed algorithms, they also show that distributed algorithms for different tasks may share some common aspects. In this paper we show that this is indeed the case for some common network control problems.

We will define the notion of a total algorithm. A total algorithm is an algorithm where the participation of all processes in the network is required before a decision can be taken. A formal definition will follow later in this section. We study the relation between total algorithms and a number of network control problems, namely Distributed Infimum computation, Resynchronization [Fi79], Propagation of Information with Feedback [Se83], Election, and Connectivity. For all these problems it will turn out that (1) any solution to these problems is necessarily total, and (2) any total algorithm can be used to solve these problems. Thus the total algorithms are a key concept in the design of network control algorithms. Many total algorithms are known. They differ in underlying assumptions about the network, in

---

The work of the author was financially supported by the Foundation for Computer Science (SION) of the Netherlands Organization for Scientific Research (NWO). The author's UUCP address is mcvax!ruuinf!gerard.

complexity, and in many other respects. We list a number of criteria for the evaluation of a total algorithm. We present a number of (known) total algorithms and compare their merits.

This paper is organized as follows. In the remainder of this section we present the model of computation and give basic definitions. The concept of total algorithms is formally defined. In section 2 we study the relation between total algorithms and some network control problems. In section 3 we give some total algorithms, together with a correctness proof. In section 4 we conclude with final comments and remaining issues.

1.1 Definitions

We consider a finite set $\mathbb{P}$ of processes, communicating only by exchanging messages. The system is asynchronous, i.e., there are no clocks and messages suffer an unpredictable delay. Message channels are point-to-point, i.e., we do not assume a bus-like communication facility like e.g. ETHERNET. Because we will consider a wide variety of algorithms and models, we make no assumptions on the topology of the network (e.g., ring, tree, complete network, etc.), except that the network is (strongly) connected. For the same reason we make no assumptions about the communication channels (e.g., FIFO, bidirectional, etc.), except that they are fault-free. That is, every message sent will be received in finite time, exactly once, and unaltered.

An execution of an algorithm consists of a sequence $a_1,...,a_k$ of *events* (cf. Lamport [La78]). The order in the sequence is by definition the order in which the events take place in the system. Of course the occurrence of these events is according to the program that the processes are running. We roughly divide the events in three classes: *send, receive* and *internal* events. In a send event a message is sent, but no message is received. In a receive event a message is received, but no message is sent. In an internal event no message is sent or received. All events can change the internal state of a process.

Send and receive events correspond naturally in a one-one way. A send event a and a receive event b correspond if the message, sent in a, is received in b. If this is the case, event a must occur earlier in the sequence of events than event b, because a message can be received only after it is sent. We define a partial order "precedes" on the events in an execution as in [La78].

Definition 1.1: Event a *precedes* event b, notation $a \rightarrow b$, if

(i) $a = b$, or a and b occur in the same process and a happens earlier than b,

(ii) a is a send, and b the corresponding receive event, or

(iii) there is an event c such that $a \rightarrow c$ and $c \rightarrow b$.

(In contrast with [La78], we defined the relation as being reflexive.) Because a message can be received only after it is sent, for any execution $a_1,...,a_k$, $a_i \rightarrow a_j$ implies $i \leq j$. For each p, let

e_p, the *enroll* event of p, be the first event that occurs in p (in a particular execution). We say p is a *starter* if e_p is a send or internal event, and p is a *follower* if e_p is a receive event.

The algorithms we consider in this paper all compute some value or bring the network in some special state. Hence we postulate the possibility of some special internal event in which a processor "decides" on the value that is to be computed or "concludes" that the network is in the desired state. The exact nature of this event is not of importance here. In the next section, where we consider some network control problems, its meaning will become clear for each problem. A decision is taken at most once in every process, and we denote the event in which it is taken in process p by d_p. Furthermore, a decision must be taken in at least one process.

Definition 1.2: An execution of an algorithm is *total* if at least one process p decides and for all $q \in \mathbb{P}$, and for all p that take a decision, $e_q \rightarrow d_p$. An algorithm is total if all its possible executions are total.

Definition 1.2 formalizes the idea that participation of all processes is required to take a decision. Finally, we say an algorithm is *centralized* if it works correct only under the assumption that there is only one starter. An algorithm is *decentralized* if it works correct, if any nonempty subset of the processes can be starters.

2 The use of total algorithms

In this section we study the relation between total algorithms and some network control problems. We use the following well-known theorem to show that algorithms for some problems are necessarily total.

Theorem 2.1: Let $a_1,...,a_k$ be an execution of some algorithm A and let $a_{\sigma(1)},...,a_{\sigma(k)}$ be a permutation of the events such that $a_{\sigma(i)} \rightarrow a_{\sigma(j)}$ implies $i \leq j$. Then $a_{\sigma(1)},...,a_{\sigma(k)}$ is a possible execution of A also.

Informally, theorem 2.1 says that any reordering of the events in an execution that is consistent with the partial ordering $\rightarrow$, is also a possible execution.

2.1 Propagation of Information with Feedback

The problem of Propagation of Information with Feedback (PIF) is explained as follows [Se83]. Starters of a PIF algorithm have a message M. All starters (if there are more than one) have the same message. This message must be broadcast, i.e., all processes must receive and accept M. The broadcast must be terminating, that is, eventually one or more processes must be notified of the completion of the broadcast. This notification is what we referred to as a "decision" in section 1.

Theorem 2.2: Every PIF algorithm is total.

Proof: Assume P is a PIF algorithm and P is non-total, i.e., there exists an execution of P where, for some process q, e_q does not precede a decision. From theorem 2.1 it follows that we can construct an execution where a decision takes place earlier than the acceptance of the message by q, so P is not correct. □

Theorem 2.3: A total algorithm can be used for PIF.

Proof: Let A be a total algorithm. Processes who want to do a broadcast act as starters in A. To every message of the execution of A, M is appended. This is possible because (1) starters of A know M by assumption and (2) followers have received a message, and thus learned M, before they first send a message. All processes q accept M in e_q. By totality of A, for all q e_q precedes any decision. Thus a decision event correctly signals the completion of the broadcast. □

To decrease the number of bits to be transmitted, we remark that it suffices to append M to the *first* message that is sent over every link.

2.2 Resynchronization

The *Resynchronization* (or, shortly, *Resynch*) problem was first described by Finn [Fi79]. It asks to bring all processes of $\mathbb{P}$ in a special state *synch* and then bring processes in a state *normal*. The state changes must be such that all processes have changed their state to *synch* before any of the processes changes state to *normal*, i.e., there is a point in time where all processes are in state *synch*. In the Resynch problem we regard the state change to *normal* as a "decision". In [Fi79] it is required that all processes change their state to *normal* eventually. We dropped this requirement. In some total algorithms only one process decides, in others all decide (see section 3). However, if not all processes decide, while it is required that all do, processes that decide can flood their decision over the network and thus force the other processes to decide also.

Theorem 2.4: Every Resynch algorithm is total.

Proof: Assume R is a Resynch algorithm and R is non-total, i.e., there exists an execution of R where, for some process q, e_q does not precede a decision. From theorem 2.1 it follows that we can construct an execution where a decision takes place earlier than the state change by q to *synch*, so R is not correct. □

Theorem 2.5: Any total algorithm can be used for Resynch.

Proof: Let A be a total algorithm. We modify A as follows. Each process q changes its state upon first participation in A, i.e., upon e_q. Each process p changes state to *normal* when it decides, i.e., upon d_p. The fact that A is a total algorithm implies the correctness of the

resulting Resynch algorithm. □

2.3 Distributed Infimum computation

Assume X is a partially ordered set with a binary infimum operator $\wedge$. That is, given x_1 and x_2 in X, it is possible to compute $y = x_1 \wedge x_2$, with the property that

(1) $y \leq x_1, y \leq x_2$,

(2) for all z, if $z \leq x_1$ and $z \leq x_2$ then $z \leq y$.

From the properties of total orderings it follows that $\wedge$ is associative, commutative, and idempotent. (On the other hand, if X is a set with a binary operator $\blacksquare$, such that $\blacksquare$ is associative, commutative, and idempotent, then there exists a partial ordering $\leq$ on X such that $\blacksquare$ is just taking infimums with regard to $\leq$.) Because of the properties of $\wedge$ it makes sense to generalize the operator to finite sets. Denote the infimum over a subset Y of X by $\inf_{x \in Y} x$. If $y = \inf_{x \in Y} x$ then

(1) for all $x \in Y, y \leq x$,

(2) if for all $x \in Y$ $z \leq x$, then $z \leq y$.

The Distributed Infimum computation problem asks for the following: suppose all processes p in $\mathbb{P}$ have a value $r_p \in X$. Compute the infimum $J = \inf_{p \in \mathbb{P}} r_p$. In the context of this problem, by a "decision" we mean the decision on the final result of the computation. In the following theorem it is assumed that X contains no smallest element, i.e., for all $x \in X$ there is a y such that $\neg\, x \leq y$.

Theorem 2.6: Every Infimum algorithm is total.

Proof: Suppose I is an Infimum algorithm and S is an execution of I where enrollment of q does not precede a decision by p. Let x be the value on which p decides. Because no participation of q precedes p's decision, we can simulate execution S up to p's decision, even if we give q a different start value r_q. Choose r_q such that $\neg\, x \leq r_q$, we now have an execution in which p decides on a wrong result. □

In most applications of infimums it is assumed that X does contain a smallest element. In this case it is possible that there are non-total executions of some Distributed Infimum algorithm. If a subset of the processes discover that the infimum over this subset is the bottom of X, they can decide that the global infimum is bottom, without consulting the other processes. However, any execution yielding another result than bottom must be total. In fact, if some Distributed Infimum algorithm contains no (hidden) tests on equality of partial results and bottom that influence the message flow or a decision, then the algorithm is necessarily total. In practice this is the case for all Distributed Infimum algorithms.

Theorem 2.7: Any total algorithm can be used as a Distributed Infimum algorithm.

Proof: Let a total algorithm A be given. Modify A as follows. Every process p is equipped with a new variable i_p, with initial value r_p. Whenever a process sends a message (as part of the execution of A) the value of i_p is attached to it. Whenever p receives a message, with i attached to it, p modifies i_p by executing $i_p := i_p \wedge i$. Internal and send events have no effects on i_p. For an event a, occurring in process p, by $i^{(a)}$ we denote the value of i_p directly after the occurrence of a. Note that if a is a send event, then $i^{(a)}$ is also the value, appended to the message sent in a. For any p, i_p is decreasing during the execution of A and thus $i^{(e_p)} \leq r_p$. Furthermore, by induction on $\rightarrow$ it is easily shown that $a \rightarrow b$ implies $i^{(a)} \leq i^{(b)}$. Thus, by the totality of A, for any decision event d, $i^{(d)} \leq i^{(e_q)} \leq r_q$ for each q. So $i^{(d)} \leq J$. It is easily seen that for all events a $i^{(a)} \geq J$. It follows that $i^{(d)} = J$, so upon a decision event in process p, p can decide on the result $i^{(d)}$. □

The problem of Distributed Infimum computation as described here should not be confused with the problem of Distributed Infimum Approximation (DIA) of [Te86]. Here we consider fixed values r_p. In the Distributed Infimum Approximation problem changing values x_p are considered. In order to let the infimum of changing values make any sense (in spite of the changes) the changes are limited in such a way that the global infimum is monotonely increasing in time. Algorithms for the DIA problem can be very complex, but a large class of them were given by Tel [Te86]. These algorithms are all based on underlying solutions for the (static) Infimum problem.

2.4 Election

For some applications it may be necessary that one process in the network is elected to perform some task or act as a "leader" in a subsequent distributed algorithm. An Election algorithm selects one process for this purpose. In the context of this problem, by a decision we mean the decision to accept some process as the leader. These decisions must be consistent, i.e., if several processes decide, they must decide on the same value. To make the problem non-trivial it is always required that a solution is decentralized and symmetric. (If it is assumed that there is only one starter, this process can immediately decide to choose itself as a leader.) By symmetric we mean that all processes are running the same local algorithm. To make a solution possible at all, it is always assumed that processes have distinct and known identification labels. For simplicity we will assume that each process p "knows" its own name p.

A large class of Election algorithms, the so-called Extrema Finding algorithms, always elect the process with the largest (or, alternatively, the smallest) identity as the leader. (It is assumed that there is a total ordering on the identities.)

Theorem 2.8: Every Extrema Finding algorithm is total.

Proof: As theorem 2.6. Now give q an identity larger than the one chosen to contradict that the chosen identity is the largest. □

Theorem 2.9: Any decentralized and symmetric total algorithm A can be used as an Election Algorithm.

Proof: As in the proof of theorem 2.7, A can be modified to yield the largest identity in the network upon a decision. To satisfy the requirement that an Election algorithm is decentralized and symmetric we assumed A to be decentralized and symmetric . □

Not all Election algorithms are Extrema Finding algorithms. Totality of a class of *comparison algorithms* can be proved, however. A comparison algorithm is an algorithm that allows comparison as the only operation on identities. For a more precise definition, see Gafni et al. [Ga84].

Theorem 2.10: Comparison Election algorithms for rings, for trees, and for general networks with unknown size are total.

Proof: We prove the result for general networks. Assume some execution S of an Election algorithm E decides on the identity of the leader in some network G without the participation of a process q in G. Now we construct a network G', consisting of two copies G_1 and G_2 of G, with one additional link q_1q_2 between the corresponding copies of q. The processes in G_2 can be renamed so that all identifiers in G' are unique, while preserving the relative ordering of identities within G_2. Because A is a symmetric comparison algorithm, the execution S can be simulated in both G_1 and G_2, leading to an execution in which decisions are taken on two values.
For rings and trees similar constructions can be given. □

The restriction to comparison algorithms mainly serves to exclude some weird algorithms without any practical use: all useful Election algorithms (also the ones mentioned below) are comparison algorithms. To show that the restriction is necessary, consider the following algorithm. First a process tests whether its identity equals 1. If this is the case, the process immediately decides and chooses itself as leader. If not, a "normal" Election algorithm is executed. This algorithm allows non-total executions if a process with identity 1 exists.

It turns out that almost all Election algorithms known to date are total. However, efforts to reduce the message complexity of Election algorithms and the search for fault-tolerant Election algorithms have led to ingenious, non-total algorithms for some special networks. One of these algorithms, notably Peterson's [Pe84], works under the assumption that the network topology is a torus. Under this assumption it is not possible to add nodes to the graph as in the proof of theorem 2.10, because the resulting graph would no longer be a torus. Another algorithm [Ba87] assumes the number of processes to be known, and again it becomes impossible to add nodes. Both algorithms are based on the idea that a majority of the processes is

found ready to accept some process as the leader. Then a decision is made for this process. Whatever happens in the uninspected parts of the network, a majority is never found there, so that inconsistent decisions are impossible.

2.5 Connectivity

The purpose of a Connectivity algorithm is to compute $\mathbb{P}$, the set of all processes. Of course it must be assumed that processes know their own identity. In fact Connectivity is a special case of the Infimum problem, because $\mathbb{P} = \underset{p \in \mathbb{P}}{\text{union}} \{p\}$. Therefore we give the following two theorems without proof:

Theorem 2.11: Every Connectivity algorithm is total.

Theorem 2.12: Any total algorithm can be used as a Connectivity algorithm.

For the latter purpose it is in general necessary to augment all messages with a list of processor identities. Messages may thus become quite long.

2.6 Traversal

The purpose of a traversal algorithm is to pass a token, initiated by a single starter, through every node in the network and return it to the starter. A traversal algorithm is total and centralized by definition. Moreover, it is *sequential*. That is, the starter sends out exactly one message in the beginning, and thereafter a process can only send (at most) one message after receiving a message. Thus, there is always at most one message in the system or one active process. Finally the starter decides, when the token returns to it after having visited all processes. Any Traversal algorithm is total by definition, but the reverse is not true because a total algorithm is not necessarily decentralized and sequential.

3 Some examples of total algorithms

This section contains a collection of total algorithms that can be found in the literature. In the literature these algorithms do not appear as a total algorithm, but they are referred to as Distributed Infimum algorithm, PIF algorithm, etcetera. Their correctness proof typically argues that some variable has the value of some (partial) infimum, or the receipt of some message implies that some subset of the processes has accepted the broadcasted message, that all processes will learn the identity of some process, etc. Because in the previous section we demonstrated that the *totality* is the key property of all these algorithms, we concentrate on this aspect only in the correctness proofs. Some algorithms will be mentioned without proof, however.

The many algorithms have varying properties, which makes some algorithms more suitable for some applications than others. We list a number of characterizing properties here.

Very important is the already mentioned distinction between centralized and decentralized algorithms. Usually decentralized algorithms are preferred because there is no need for a special process to start the execution. Unfortunately their complexity is often larger. For a ring (of N processes) there is a centralized algorithm (algorithm A) that uses N messages, which is (order) optimal, whereas for decentralized algorithms on a ring $O(N \log N)$ is optimal (algorithm H). For a general network (of N processes and E edges) there is a centralized algorithm using $2E$ messages (algorithm C), which is (order) optimal, whereas for decentralized algorithms on general networks $O(N \log N + E)$ is optimal (algorithm G). Decentralization seems to cost $O(N \log N)$ messages. Surprisingly, the complexities of centralized and decentralized algorithms for trees are the same.

We say that a distributed algorithm is *symmetric* if the local algorithm is the same for each process. The notion of symmetry is closely related to that of being or not being decentralized. In most centralized algorithms the starter runs a local algorithm, different from that of the followers, and most decentralized algorithms are symmetric. Algorithm B below is symmetric, but not decentralized.

Algorithms are often designed for a special network topology, such as a ring, a tree, a complete network, or for general networks. Assumptions about the communication channels can also be made, such as that they are bidirectional, obey the FIFO discipline, or provide synchronous communication. Sometimes it is assumed that processes have some knowledge about the network, such as its diameter, or the names of their neighbors.

In some algorithms all processes decide, in others only one or few. In the original statement of some problems, e.g., Resynchronization [Fi79], it is explicitly required that all processes decide. If, in some total algorithm, not all processes decide, the deciding process(es) can flood a signal over the network to make other processes decide also. Therefore this aspect of total algorithms is of minor importance only.

The message and time complexity, and internal storage used in a process are quantitative aspects for the evaluation of an algorithm.

Algorithm A: A centralized total algorithm for a (unidirectional) ring. The starter sends out a token on the ring, which is passed by all processes and finally returns to the starter. Then the starter decides. Assume each process p has a boolean variable Rec_p to indicate whether a message has been received. Initially its value is false. Let $\langle\rangle$ denote an empty message. As usual, the sentence between braces is a guard which must be true in order for the event to execute. The program for the starter is:

S_p: { Spontaneous start, execute only once }
send <> to successor

R_p: { A message <> arrives }
begin *receive* <> ; Rec_p := *true* **end**

D_p: { Rec_p }
decide

and for all other processes:

R_p: { A message <> arrives }
begin *receive* <> ; Rec_p := *true* **end**

S_p: { Rec_p }
send <> to successor

For our analysis, assume the processes are numbered in such a way that process i+1 is the successor of process i (indices are counted modulo N). Call f_i the event in which i sends, g_i the event in which i receives, d_i the event in which i decides (if this happens), and l the starter. We have $e_l = f_l$, and $e_i = g_i$ for other i. From the algorithm text we have $g_i \rightarrow f_i$ if i is not l, and we have $f_i \rightarrow g_{i+1}$ because these are corresponding events. Only the starter decides, and we have $g_l \rightarrow d_l$ directly from the algorithm text. Thus, using the fact that there is only one starter, we find

$$e_l = f_l \rightarrow g_{l+1} = e_{l+1} \rightarrow f_{l+1} \ldots \rightarrow g_i = e_i \rightarrow f_i \ldots \rightarrow g_l \rightarrow d_l,$$

which establishes the totality of A. The fact that a decision is at all taken in algorithm A is trivial.

Our framework allows us to analyze the behavior of the algorithm in case two or more processes act as a starter. Suppose l_1 and l_2 act as starters. Then we find

$$e_{l_1} = f_{l_1} \rightarrow g_{l_1+1} = e_{l_1+1} \rightarrow f_{l_1+1} \ldots \rightarrow g_i = e_i \rightarrow f_i \ldots \rightarrow g_{l_2} \rightarrow d_{l_2}$$

and

$$e_{l_2} = f_{l_2} \rightarrow g_{l_2+1} = e_{l_2+1} \rightarrow f_{l_2+1} \ldots \rightarrow g_i = e_i \rightarrow f_i \ldots \rightarrow g_{l_1} \rightarrow d_{l_1},$$

but clearly the execution is not necessarily total.

Algorithm A is a centralized algorithm for a (unidirectional) ring. No FIFO discipline on links or knowledge about the number of nodes is assumed. It is not required that processes

have identities. The message and time complexity are both N, and one bit of internal storage suffices.

Algorithm B: A total algorithm for a tree network. Note that a tree network must have bidirectional links in order to be strongly connected. A process that has received a message over all links except one sends a message over this last link. Leaves of the tree have only one link, and thus, they must always be starters. A process that has received a message over all links decides. In the following formal description of the algorithm, $Neigh_p$ is the set of neighbors of p, and p has a boolean variable $Rec_p[q]$ for each $q \in Neigh_p$. The value of $Rec_p[q]$ is *false* initially. Although this is not explicit in the following description, it is intended that each process sends only once. Thus, a send action by p disables further send actions in p.

$\mathbf{R}_p$: { A message $\langle\rangle$ from q arrives at p }
begin *receive* $\langle\rangle$; $Rec_p[q] := true$ **end**

$\mathbf{S}_p$: { $q \in Neigh_p$ is such that $\forall r \in Neigh_p, r \neq q : Rec_p[q]$ }
begin *send* $\langle\rangle$ to q **end**

$\mathbf{D}_p$: { for all $q \in Neigh_p : Rec_p[q]$ }
begin *decide* **end**

In the following analysis, let f_{pq} be the event that p sends a message to q, g_{pq} the event that q receives this message, and d_p the event that p decides. Let T_{pq} be the subset of the nodes that are reachable from p without crossing the edge pq (if this edge exists). By the connectivity of the network we have

$$T_{pq} = \{p\} \cup \operatorname*{union}_{r \in Neigh_p - \{q\}} T_{rp} \tag{T1}$$

and

$$\mathbf{P} = \{p\} \cup \operatorname*{union}_{r \in Neigh_p} T_{rp}. \tag{T2}$$

Lemma 3.1: For all $s \in T_{pq}$, $e_s \rightarrow g_{pq}$.

Proof: By induction on $\rightarrow$. Assume the lemma is true for all receive actions that precede g_{pq}. Let $s \in T_{pq}$. By T1, $s = p$ or $s \in T_{rp}$ for some $r \in Neigh_p$, $r \neq q$. We have $f_{pq} \rightarrow g_{pq}$ because these events correspond, $e_p \rightarrow f_{pq}$ because e_p precedes all events in p, so $e_p \rightarrow g_{pq}$ follows. By virtue of the algorithm $g_{rp} \rightarrow f_{pq}$ for all neighbors $r \neq q$ of p, and by the induction hypothesis $e_s \rightarrow g_{rp}$ for all s in T_{rp}. So $e_s \rightarrow g_{pq}$ follows. □

Theorem 3.2: For all $s \in \mathbb{P}$, $e_s \rightarrow d_p$.

Proof: By T2, for all $s \in \mathbb{P}$, $s = p$ or $s \in T_{rp}$ for some $r \in Neigh_p$. We have $e_p \rightarrow d_p$ as above. If $s \in T_{rp}$, then we have $g_{rp} \rightarrow d_p$ by the algorithm, $e_s \rightarrow g_{rp}$ by the previous lemma, and $e_s \rightarrow d_p$ follows. □

In contrast with algorithm A, in this case it is not obvious that a decision is reached at all. We will show by a simple counting argument that this is the case.

Theorem 3.3: Assume that all events of algorithm B that are enabled will eventually be executed. Then a decision is eventually taken.

Proof: We first show that as long as no decision is taken there is always a next event enabled to be executed. There are $2E = 2(N-1)$ *Rec*-bits in the network. Define, for a certain system state, F to be the number of *Rec* bits that are false, F_p the number of these in process p, K the number of processes that have sent, and M the number of messages underway. Observe $F = 2N - 2 + M - K$. If $M > 0$, there is a message underway and eventually a receive event will occur. If $M = 0$, then $F = 2N - 2 - K < 2(N-K) + K$. It follows that (1) under the $N - K$ processes that have not yet sent there is a process p with $F_p < 2$ or (2) under the K processes that have sent there is a process with $F_p < 1$. In case (1) this process will eventually send, in case (2) this process will eventually decide. Thus, while no process has decided, there is always a next send, receive, or decide event that will eventually take place. But then, because the total number of send actions is bounded by N (each process sends at most once), it follows that a decision will be taken in finite time. □

Because all messages will be received and all processes send exactly once, a state is reached where $K = N$ and $M = 0$, thus $F = N - 2$. It follows that exactly two processes decide. It turns out that these two processes are neighbors.

Again, our framework allows us to analyze the behavior of the algorithm if it is used on a network that is not a tree. Theorem 3.2 remains true (its proof relies on identities T1 and T2, and these follow from the connectivity of the network only), but theorem 3.3 fails (its proof relies on the fact that the number of edges is $N - 1$). In fact, it is easily seen that if the network contains a cycle, no process on this cycle will ever send. Thus no decision will be taken.

Algorithm B works on a tree network with bidirectional links. The processes need not have distinct identities, it is enough that a process can distinguish between its links. The algorithm is simple and symmetric. Its message complexity is N, its time complexity is D (the diameter of the tree). The internal storage in a process is a number of bits of the order of its degree in the network.

For the assumption in theorem 3.3 it is necessary that all leaves (except possibly one) are starters. Hence not every nonempty subset of the processes suffices to start the algorithm, and the algorithm is not decentralized. Algorithm B can easily be transformed to make it

decentralized: starters flood a "wake-up" signal over the tree to trigger execution of the algorithm is every process. Assuming that nodes relay the wake-up message to every neighbor except the one they received the signal from, the number of wake-up messages is $(N-2)+s$, where s is the number of starters. For starters that are leaves the wake-up message can be combined with the message of the algorithm, so that only $(N-2)+s'$ new message are necessary, where s' is the number of starters that is not a leaf. This number of messages must be added to the message complexity N of the given version of algorithm B.

Algorithm C: A centralized total algorithm for general bidirectional networks. This algorithm is usually referred to as Chang's Echo algorithm [Ch82]. The starter sends a message over all links. Followers remember the link over which they first received a message as their *father*. Upon receiving their first message followers send a message to all neighbors except their father. When a follower has received a message over all links it sends a message to its father. When the starter has received a message over all links it decides. In the following formal description of the algorithm, let Rec_p and $Neigh_p$ be as in the description of algorithm B. In the program fragment labeled with S_p several messages can be sent. This program fragment describes a sequence of events, to be executed by the process, rather than a single event. We use this notation for brevety. The same remarks apply to R_p below and later fragments. The program for the starter is:

S_p: (* Spontaneous start, execute only once *)
forall $q \in Neigh_p$ **do** *send* $\langle\rangle$ to q

R_p: { A message $\langle\rangle$ arrives from q }
begin *receive* $\langle\rangle$; $Rec_p[q] := true$ **end**

D_p: { $\forall q \in Neigh_p : Rec_p[q]$ }
decide

The program for a follower is:

$\mathbf{R}_p$: { A message $\langle\rangle$ arrives from q }
begin *receive* $\langle\rangle$; $Rec_p[q] := true$;
if $father_p = nil$ **then**
begin $father_p := q$;
forall $r \in Neigh_p - \{q\}$ **do** *send* $\langle\rangle$ to r
end
end

$\mathbf{S}_p$: { $\forall q \in Neigh_p : Rec_p[q]$ }
send $\langle\rangle$ to $father_p$

It turns out that algorithm C builds a spanning tree in the network, and than works as algorithm B on this (rooted) tree. The following correctness proof is found in [Se83]. Consider a complete execution S of algorithm C and observe that the $father$ fields, once given a value $\neq nil$, are never changed thereafter. Define a directed graph T, consisting of the processes as nodes and all links from p to $father_p$ (for all p with $father_p \neq nil$ at the end of S) as edges.

Lemma 3.4: T is a rooted tree with the starter as root.

Proof: Observe that (1) each node of T has at most one outgoing edge, (2) T is cycle free (because $e_{father_p} \rightarrow e_p$ is easily derived from the algorithm), and (3) each non-starter has an outgoing edge (because, if a process p sends to its neighbor q, q eventually receives this message and then (i) q is starter, (ii) q assigns the value p to $father_q$, or (iii) $father_q$ had a value already). Because there is only one starter the result follows. □

Define T_p to be the subtree under p, let d_p, f_{pq}, and g_{pq} be as before, and let l be the starter.

Theorem 3.5: For all q, $e_q \rightarrow d_l$.

Proof: As in the proof of lemma 3.1 it can be shown that if rp is an edge, then for all q in T_r $e_r \rightarrow g_{rp}$. As in the proof of theorem 3.2 the result follows. □

By a counting argument similar to the argument in theorem 3.3 it is shown that the starter will decide indeed.

Our framework allows us to analyze the behavior of the algorithm in case two or more processes act as a starter. Then T will not be a rooted tree, but rather a rooted forest. Each starter is the root of one tree. The decision of a starter is preceded by the enroll events of the processes in the tree of which it is the root, and their neighbors. As in the case of algorithm A, the execution is not necessarily total.

Algorithm C works for bidirectional networks of arbitrary topology. It is centralized, and the processes need not have identities. The algorithm is simple. Its message complexity is $2E$, its time complexity is D.

Algorithm D: A decentralized total algorithm for general directed networks with known diameter D. Each process sends D times a message to all of its out-neighbors. It sends the $i+1^{\text{th}}$ message only after receiving i messages from all of its in-neighbors. A process that has received D messages from all of its in-neighbors decides. A more precise description follows. Let for each process p In_p be the set of its in-neighbors, Out_p the set of its out-neighbors, and assume p has a counter $RCount_p[q]$ for each $q \in In_p$, and a counter $SCount_p$. Initially all counters are 0.

$\mathbf{R}_p$: { A message $\langle\rangle$ arrives from q }
begin *receive* $\langle\rangle$; $RCount_p[q] := RCount_p[q]+1$ **end**

$\mathbf{S}_p$: $\{ \forall q \in In_p: RCount_p[q] \geq SCount_p \ \wedge \ SCount_p < D \}$
begin forall $r \in Out_p$ **do** *send* $\langle\rangle$ to r ;
$SCount_p := SCount_p + 1$
end

$\mathbf{D}_p$: $\{ \forall q \in In_p: RCount_p[q] \geq D \}$
decide

In this algorithm more than one message can be sent over a link. Let, if an edge pq exists, $f_{pq}^{(i)}$ be the i^{th} event in which p sends to q, and $g_{pq}^{(i)}$ be the i^{th} event in which q receives from p. If a FIFO discipline on links is assumed these events correspond, so that trivially $f_{pq}^{(i)} \rightarrow g_{pq}^{(i)}$. We do not assume a FIFO discipline so that the following result becomes non-trivial.

Theorem 3.6: $f_{pq}^{(i)} \rightarrow g_{pq}^{(i)}$.

Proof: Define m_h such that $f_{pq}^{(m_h)}$ corresponds with $g_{pq}^{(h)}$, i.e., in its h^{th} receive event q receives p's m_h^{th} message. We have $f_{pq}^{(m_h)} \rightarrow g_{pq}^{(h)}$. Each message is received only once, so all m_h are different. This implies that at least one of $m_1,...,m_i$ is greater than or equal to i. Let $m_j \geq i$, then $f_{pq}^{(i)} \rightarrow f_{pq}^{(m_j)} \rightarrow g_{pq}^{(j)} \rightarrow g_{pq}^{(i)}$. □

The truth of this lemma is not restricted to this particular case of algorithm D. It is an important result, which can be used in the analysis of any algorithm for systems with non-FIFO channels. It holds even in the case messages may get lost in the links. The only assumption that must be made about the links is that every message is received only once, that is, that no duplication of messages occurs. We continue the correctness proof of algorithm D.

Theorem 3.7: For all $s \in \mathbb{P}$, $e_s \rightarrow d_p$.

Proof: Let $p_0 p_1 .. p_l$, $l \leq D$, be a path in the network. By the previous theorem we have $f^{(i+1)}_{p_i p_{i+1}} \rightarrow g^{(i+1)}_{p_i p_{i+1}}$ for $i < l$ and by the algorithm we have $g^{(i+1)}_{p_i p_{i+1}} \rightarrow f^{(i+2)}_{p_{i+1} p_{i+2}}$ for $i < l - 1$. Thus $e_{p_0} \rightarrow g^{(l)}_{p_{l-1} p_l}$. Because the network diameter is D, for every s and p there is a path $s = p_0 .. p_l = p$ of length at most D. Thus $e_s \rightarrow g^{(l)}_{p_{l-1} p}$. By the algorithm $g^{(l)}_{p_{l-1} p} \rightarrow d_p$, so that the result follows. □

Theorem 3.8: Assume that all events of algorithm D that are enabled will eventually be executed. Then all processes will decide.

Proof: First we show that, while not all processes have decided, there is always a next event enabled to execute. If there are messages underway, a receive event will be enabled. Suppose there are no message underway, note that this implies $RCount_q[r] = SCount_r$ if rq is an edge. Let S be the smallest of all $SCount$ registers in processes, and p be such that $SCount_p = S$. If $S = D$ a decision is enabled in all processes. If $S < D$ then for all $q \in In_p$, $RCount_p[q] = SCount_q \geq SCount_p$ and $SCount_p < D$, so a send event is enabled in p. Thus algorithm D does not deadlock until all processes have decided. Furthermore, only a finite number of events can take place, namely $D\,E$ send, $D\,E$ receive, and N decide events. It follows that all processes will decide. □

In [Te87] it is shown that in order to avoid deadlock it is sufficient that at least one process is a starter. In [Te87] more properties of this algorithm are proved and it explicitly shows its use as PIF algorithm, Distributed Infimum algorithm, Resynch algorithm, etc. The behavior of the algorithm on networks that are not strongly connected is also analyzed there.

Algorithm D is decentralized and works on any (directed) network. It is required that (an upper bound for) the network diameter is known. The processes need not have distinct identities. The message complexity is $D\,E$, the time complexity of algorithm D is D.

Algorithm E: Another decentralized algorithm for general unidirectional networks. Each process p maintains two sets of processor identities. Informally speaking, HO_p is the set of processes p has heard of, and OK_p is the set of processes such that p has heard of all the in-neighbors of these processes. Initially $HO_p = \{p\}$, and $OK_p = \emptyset$. HO_p and OK_p are included in every message p sends. Upon receiving a message (containing a HO and an OK set), p replaces HO_p and OK_p by the union of the old and the received version of the set. When p has received a message from all of its in-neighbors, p inserts its own identity p in OK_p. When $HO_p = OK_p$, p decides. A formal description of the algorithm follows. Let In_p and Out_p be as for algorithm D, and $Rec_p[q]$ as for algorithm C.

S_p: *send* $<HO_p, OK_p>$ to some $q \in Out_p$

$\mathbf{R}_p$: { A message $<HO, OK>$ from q arrives at p }
begin *receive* $<HO, OK>$; $Rec_p[q] := true$;
$HO_p := HO_p \cup HO$; $OK_p := OK_p \cup OK$
end

$\mathbf{A}_p$: { $\forall q \in In_p : Rec_p[q]$ }
$OK_p := OK_p \cup \{p\}$

$\mathbf{D}_p$: { $HO_p = OK_p$ }
decide

Let a_p denote the (first) execution of the A event in process p. For any event b (occurring in process p), let $HO^{(b)}$ and $OK^{(b)}$ denote the value of HO_p and OK_p immediately after the occurrence of b.

Lemma 3.9: If $e_q \rightarrow b$ then $q \in HO^{(b)}$.

Proof: The updates of the HO sets in processes and messages imply, as in the proof of theorem 2.7, that for any two events b and c, $c \rightarrow b$ implies $HO^{(c)} \subseteq HO^{(b)}$. By the algorithm $q \in HO^{(e_q)}$, so the result follows. □

Lemma 3.10: If $q \in OK^{(b)}$ then for all $r \in In_q$, $e_r \rightarrow b$.

Proof: First we prove by induction on $\rightarrow$ that $q \in OK^{(b)}$ implies $a_q \rightarrow b$. Let p be the process where b occurs. If b is a send or decide event, OK_p does not change in the event so an earlier event b' in p must have caused q to be in OK_p. Use induction and $b' \rightarrow b$ to conclude the result. If b is a_p then $p = q$ or q was in OK_p already before the occurrence of b. In the first case the result is trivial, in the second case it follows as for send and decide events. If b is a receive event then q was in the OK set of the message that is received, or q was in OK_p already before the occurrence of b. In the first case, use the induction hypothesis for the corresponding send event, in the second case the result follows as above. Thus $q \in OK^{(b)}$ implies $a_q \rightarrow b$. From the algorithm it follows that for all $r \in In_q$, $e_r \rightarrow a_q$, from which the result follows immediately. □

Theorem 3.11: For all r, $e_r \rightarrow d_p$.

Proof: Trivially $e_p \rightarrow d_p$. If q is a process such that $e_q \rightarrow d_p$, then by 3.9 $q \in HO^{(d_p)}$, so by $HO^{(d_p)} = OK^{(d_p)}$ we have $q \in OK^{(d_p)}$, and by 3.10 we find that for all $r \in In_q$, $e_r \rightarrow d_p$. The result follows from the strong connectivity of the network. □

The message complexity of algorithm E as given is unbounded, because a send event can in principle be repeated infinitely often. We can restrict sending of messages by p in such a way

that sending is allowed only if OK_p or HO_p has a value that was not sent to the same process before. In that case the message complexity is bounded by $2NE$ messages. Under this restricted sending policy we can prove:

Theorem 3.12: Assume all events of algorithm E that are enabled will eventually be executed. Then all processes will decide.

Proof: We first show that, while not all processes have decided, there is always a next event enabled to execute. If there is a link pq such that the current value of HO_p and OK_p has not been sent over it, a send event is enabled. Assume (1) that for each link pq the current value of HO_p and OK_p has been sent over it. If there is a link pq such that this value has not been received by q, a receive event is enabled. Assume (2) that for each link this value has been received. Then for each q the addition of q to OK_q is enabled. Assume (3) that this addition has taken place for all q. From assumption (2) follows that $HO_p \subseteq HO_q$, thus by the strong connectivity of the network, all HO are equal. From $r \in HO_r$ for all r follows $HO_p = \mathbb{P}$ for all p. From assumption (2) we can also derive that all OK sets are equal and, using assumption (3), that $OK_p = \mathbb{P}$ for all p. But then for all p $OK_p = HO_p$, and a decision is enabled in all processes. Thus the system does not deadlock before all processes have decided. Because only $2NE$ send, $2NE$ receive, N addition, and N decision events are possible, it follows that all processes will decide. □

Algorithm E is in fact Finn's Resynch algorithm [Fi79]. To see this, first observe that always $OK \subseteq HO$ for all processes and messages. Thus, the two sets can be represented by a vector as follows. In this vector there is an entry for each potential member of $\mathbb{P}$. The entry can be 0, 1, or 2. A 0 entry means the process is neither in HO nor in OK, a 1 entry means the process is in HO but not in OK, a 2 means that the process is in both HO and OK. The test $OK = HO$ now reads: the vector contains no 1's. Two differences between this algorithm and Finn's are important to mention. First, Finn assumes bidirectional links, where we assume strong connectivity of the network. If links are assumed bidirectional, (weakly) connected components of the network are strongly connected. Finn uses the algorithm to determine the nodes in one component and synchronize this component only. A second difference is that Finn's algorithm also provides a mechanism to restart the algorithm after a topological change. This, now fairly standard, mechanism was described separately by Segall [Se83].

A consequence of the material in this paper is that the Resynch problem [Fi79] can be solved by an algorithm using fewer messages. In Finn's work bidirectional channels are assumed, so algorithm E can be replaced by e.g. algorithm G. For the resynchronization of unidirectional networks the algorithm of Gafni and Afek [GA84] can be used instead. An advantage of algorithm E over the others is its low time complexity.

Algorithm E is decentralized and works on any (directed) network. Processes must have distinct identities. The message complexity is (at most) $O(NE)$ messages (of size N

identities) and the time complexity is D.

It is interesting to compare algorithms B, C, D, and E. Algorithm B assumes a tree topology, algorithm C assumes there is exactly one starter, algorithm D assumes the network diameter to be known, and algorithm E assumes processes have identities. It can be shown that at least one of these assumptions is necessary: there exists no decentralized algorithm for anonymous, general networks with no bound on the diameter.

To give the reader an impression of the wealth of known total algorithms we mention a few more of them, without precise description or correctness proof.

Algorithm F: Distributed Depth First Search (DDFS) [Ch83]. DDFS is a centralized algorithm for general bidirectional networks. It is sequential, i.e., DDFS is a Traversal algorithm. Both its message and time complexity are $2E$. The processes need not have distinct identities. Intricate variants with an $O(N)$ time complexity exist (see e.g. Awerbuch [Awe85]), but these variants are no longer sequential. A version for directed networks exists [GA84].

Algorithm G: Gallager, Humblet, and Spira's algorithm for distributed Minimum Spanning Tree construction [GHS83]. This is a decentralized algorithm for general bidirectional networks. Distinct identities are required. The message complexity is $O(N\log N + E)$, which is provably optimal, and its time complexity is $O(N\log N)$. Noteworthy is the adaption by Gafni and Afek to directed networks [GA84].

Algorithm H: Many algorithms have been given for Election on (unidirectional) rings. Their complexity is typically $O(N\log N)$. Peterson's algorithm [Pe82] is noteworthy because it runs on unidirectional rings and has a message complexity of $O(N\log s)$, where s is the number of starters.

Of course this list can be made much longer. Only a few of the most important algorithms are mentioned here.

4 Remaining issues

We discuss some remaining issues and relations with other problems.

4.1 The Gossip Exchange problem

The Gossip Exchange problem is described as follows. Each of N uncles $u_1,...,u_N$ knows a story. How many telephone calls are necessary to reach the situation where each of the uncles knows each of the N stories? It is assumed that every uncle can make calls to every other uncle, that one call connects precisely two uncles and that no uncle can be engaged in two calls simultaneously, and that during a telephone call the two participating uncles exchange all information they have. The problem is originally due to A. Boyd.

It is not difficult to see that $2N - 4$ call suffice, and it was proven by Tijdeman that this is optimal [Tij71]. In our terminology, the problem asks for a total algorithm where each process decides, in a certain model of communication. This model is characterized by (1) a complete communication network, (2) communication by "rendez-vouz" instead of asynchronous messages, and (3) a synchronous execution model. The work of [Tij71] shows that under this model a communication complexity of $2N - 4$ is optimal for total algorithms.

This paper shows that any solution to the Gossip Exchange can also be used to broadcast a message under the uncles, to compute some infimum over the uncles, or to "resynchronize" them.

4.2 Constructions of decentralized total algorithms from centralized ones

In the usual statement of the Election problem it is required that a solution is decentralized. In this section we give some constructions that build decentralized total algorithms from centralized ones: Extinction (see below), Korach et al.'s construction, and Attiya's construction. In all constructions, starting an election is done by starting a copy of the centralized algorithm, in which messages are tagged with the initiator's identity. The copies run concurrently and compete, until finally a decision is taken in one of them. The initiator of this copy is then elected.

Construction 1: (Simple Extinction) We construct a decentralized total algorithm E from a centralized total algorithm C. A starter p of E starts a copy C_p of C, with all messages tagged with p. Each p maintains m_p, the highest identity p has seen so far (initially p). On receipt of a message from C, tagged with $q < m_p$, p will start a copy of C (unless it did so earlier), but not participate in C_q, i.e., it will not execute any event of the execution C_q. If $q \geq m_p$, p will (eventually) update m_p and participate in C_q. If a decision is taken in C_q, this is regarded as a decision in E.

Let m be the highest identity of any process. By construction, only in C_m a decision can be taken, and C_m will start eventually. If the message and time complexity of C are M and T, respectively, then the message and time complexity of E are bounded by $O(NM)$ and $O(NT)$, respectively. The reader is invited to construct an example where the complexity of E actually meets these bounds.

The Extinction construction can be improved as follows. A non-starter, receiving a message, will not purge it, nor start its own C, and will not start one thereafter. Let M and T be as before, and s be the number of starters. Then the message and time complexity of the constructed algorithm are now $O(sM)$ and $O(sT)$, respectively. The algorithms of Chang and Roberts [CR79] and Hirschberg [Hi80] are obtained by applying Extinction to Algorithms A and C, respectively.

Construction 2: (Korach et al., [KKM85]) Starting from a traversal algorithm with message complexity M, Korach et al. construct an Election algorithm with message complexity $O((M+N)\log s)$.

Construction 3: (Attiya, [At87]) Starting from a Traversal algorithm with message complexity $f(N)$, Attiya constructs an Election algorithm for undirected networks with a message complexity of $\sum_{k=1}^{N} f(\frac{N}{k})$.

Constructions 2 and 3 use fewer messages than construction 1, but they are less general, for they require a traversal algorithm, whereas construction 1 can use a centralized, but not sequential, total algorithm.

4.3 Repeated execution

In many applications of total algorithms repeated execution of the algorithm is necessary. See for example the Distributed Selection algorithm of Santoro et al. [SSS87], or the construction of Distributed Infimum Approximation algorithms from Infimum algorithms [Te86]. In these applications it is required that subsequent executions of the algorithm are *disjoint*. Let A be a total algorithm which is executed repeatedly, and $e_p^{(i)}$ denote p's enroll event in the i^{th} execution of A.

Definition 4.1: A series of executions of A is called *disjoint* if for all p, q, and i: $e_p^{(i)} \rightarrow e_q^{(i+1)}$.

An easy way to achieve disjointness is by adopting the following rule:

Rule: A process may act as a starter in execution $i+1$ only if it has decided in execution i.

In any execution of A, for all q there is a p such that p is a starter and $e_p \rightarrow e_q$. This fact, together with the rule, implies indeed that the subsequent executions are disjoint.

Centralized algorithms in which only the starter decides (algorithms A, C, and F from section 3) require that once a starter is elected. This process will be the starter of all subsequent executions.

Decentralized algorithms, in which any number of processes can start (algorithms D, E, G, and H), do not require such an election. If only one process decides in the execution, and

the message complexity is small if there are few starters, all executions except the first have low message complexity. For example, the first execution of algorithm H takes $O(N\log N)$ messages, all subsequent executions take N messages. This makes algorithm H particularly interesting for repeated execution. The same holds for decentralized algorithms that are obtained by applying the improved Extinction construction to some centralized algorithm.

Algorithm B poses a special problem if it must be executed repeatedly. Here the leaves of the tree must be starters, but they do not necessarily decide. A deadlock results. This problem can be remedied by adapting the algorithm as follows. A process that decides broadcasts a message over the tree and all processes decide also on receipt of this message. Then the leaves are ready to start a next execution.

5 References

[At87] Attiya, H., *Constructing Efficient Election Algorithms from Efficient Traversal Algorithms*, in: J. van Leeuwen (ed.), Proceedings 2nd International Workshop on Distributed Algorithms, Lecture Notes in Computer Science 312, Springer Verlag, 1988.

[Awe85] Awerbuch, B., *A New Distributed Depth First Search Algorithm*, Inf. Proc. Lett. 20 (1985) 147-150.

[Ba87] Bar–Yehuda, R., S. Kutten, Y. Wolfstahl, S. Zaks, *Making Distributed Spanning Tree Algorithms Fault Resilient*, in: F.J. Brandenburg et al. (ed.), Proceedings STACS87, LNCS 247, Springer Verlag, Heidelberg, 1987.

[Ch82] Chang, E.J.H., *Echo Algorithms: Depth Parallel Operations on General Graphs*, IEEE Trans. Software Eng. SE-8 (1982) 391-401.

[Ch83] Cheung, T., *Graph Traversal Techniques and the Maximum Flow Problem in Distributed Computation*, IEEE Trans. Software Eng. SE-9 (1983) 504-512.

[CR79] Chang, E., R. Roberts, *An Improved Algorithm for Decentralized Extreme Finding in Circular Arrangements of Processes*, Comm. ACM 22 (1979), 281–283.

[Fi79] Finn, S.G., *Resynch Procedures and a Fail-safe Network Protocol*, IEEE Trans. Communications COM-27 (June 1979), 840-845.

[GA84] Gafni, E., Y. Afek, *Election and Traversal in Unidirectional Networks*, Proc. 4th symp. on Principles of Distr. Comp., Vancouver, Canada, 1984.

[Ga84] Gafni, E., M.C. Loui, P. Tiwari, D.B. West, S. Zaks, *Lower Bounds on Common Knowledge in Distributed Systems*, Techn. Rep R–1017, Coordinated Science

Laboratory, University of Illinois, September 1984.

[Ga86] Gafni, E., *Perspectives on Distributed Network Protocols: a Case for Building Blocks*, Proceedings IEEE Military Communications Conference, Montery, October 1986.

[GHS83] Gallager, R.G., P.A. Humblet, P.M. Spira, *A Distributed Algorithm for Minimum-Weight Spanning Trees*, ACM ToPLaS 5 (1983), 67-77.

[Hi80] Hirschberg, D.S., *Election Processes in Distributed Systems*, Dept. of Comp. Sc., Rice Univ., Houston, Texas, 1980.

[KKM85] Korach, E., S. Kutten, S. Moran, *A Modular Technique for the Design of Efficient Distributed Leader Finding Algorithms*, Proceedings 4th Symp. on Principles of Distr. Computing, 1985.

[La78] Lamport, L., *Time, Clocks, and the Ordering of Events in a Distributed System*, Comm. ACM 21 (1978), 558-565.

[Pe82] Peterson, G.L., *An O(nlogn) Unidirectional Algorithm for the Circular Extrema Problem*, ACM ToPLaS 4 (1982), 758-762.

[Pe84] Peterson, G.L., *Efficient Algorithms for Election in Meshes and Complete Networks*, TR-140, University of Rochester, Rochester, Aug. 1984.

[Se83] Segall, A., *Distributed Network Protocols*, IEEE Trans. Inf. Theory IT-29 (1983), 23-35.

[SSS87] Santoro, N., J.B. Sidney, S.J. Sidney, *On the Expected Complexity of Distributed Selection*, in: F.J. Brandenburg et al. (ed.), Proceedings STACS87, LNCS 247, Springer Verlag, Heidelberg, 1987, pp. 456-467.

[Te86] Tel, G., *Distributed Infimum Approximation*, Techn. Rep. RUU-CS-86-12, Dept. of Computer Science, University of Utrecht, Utrecht, 1986.

[Te87] Tel, G., *Directed Network Protocols*, in: J. van Leeuwen (ed.), Proceedings 2nd International Workshop on Distributed Algorithms, Lecture Notes in Computer Science 312, Springer Verlag, 1988.

[Tij71] Tijdeman, R., *On a Telephone Problem*, Nieuw Arch. v. Wisk. XIX (1971) 188-192.

[TL86] Tan, R.B., J. van Leeuwen, *General Symmetric Distributed Termination Detection*, Techn. Rep. RUU-CS-86-2, Dept. of Computer Science, University of Utrecht, Utrecht, 1986.