Total Algorithms[*]

Gerard Tel[†]
Department of Computer Science, University of Utrecht,
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands.
email: `gerard@cs.ruu.nl`

April 1988
Revised November 1989 and February 1993

Abstract

We define the notion of *total algorithms* for networks of processes. A total algorithm enforces that a "decision" is taken by a subset of the processes, and that participation of all processes is required to reach this decision. Total algorithms are an important building block in the design of distributed algorithms. For some important 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 for a variety of network topologies. Constructions are shown to derive algorithms for Mutual Exclusion, Election, and Distributed Infimum Approximation from arbitrary total algorithms. The paper puts many results and paradigms about designing distributed algorithms in a general framework.

This report outlines several other works of the author. Total algorithms, their properties, and some additional examples, as well as traversal algorithms and the time complexity of distributed algorithms are studied in [Tel94, Chap. 6]. The construction of algorithms for distributed infimum approximation is treated in [CBT94, Tel86] and [Tel91, Sec. 4.1].

Key Words: Distrubuted Algorithms, Design methods, Broadcast, Mutual Exclusion, Election, Termination Detection.

1 Introduction

Modular techniques have been advocated recently to facilitate the design and verification of distributed algorithms. Noteworthy examples are Gafni [Gaf86], Segall [Seg83], and Korach *et al.* [KKM90]. 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 for *asynchronous* networks.

[*]This report has appeared (1) as Technical Report RUU–CS–88–16; (2) in M. Cosnard (ed.), *Parallel and Distributed Algorithms*, Elseviers, 1989; (3) in F.H. Vogt (ed.), *Concurrency88*, LNCS 335, pp. 353–367, 1988; (4) in Algorithms Review 1 (1990) 13–42.

[†]The work of this author was supported by ESPRIT Basic Research Action No. 7141 (project ALCOM II: *Algorithms and Complexity*).

We will define a total algorithm as an algorithm where the participation of all processes in the network is required before a decision is 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 [Fin79], Propagation of Information with Feedback [Seg83], Extrema Finding, 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. We will also see that algorithms for Mutual Exclusion and for Distributed Infimum Approximation can be constructed in a generic way, based on arbitrary total algorithms. The problem of Distributed Infimum Approximation (defined in [Tel86]) includes both the Termination Detection problem and approximation of Global Virtual Time. Thus the total algorithms are a key concept in the design of network control algorithms. We present a number of total algorithms and some constructions of other algorithms based upon them.

This paper is organized as follows. First we present a definition of totality of an algorithm. In section 2 we study the relation between total algorithms and a number of network control problems. In section 3 we give some total algorithms and their correctness proof. In section 4 we consider the problem of Mutual Exclusion in distributed systems. In section 5 we consider the problem of Distributed Infimum Approximation. In section 6 some more remarks about total algorithms are made.

1.1 Definition of Total Algorithms

As usual 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 finite delay. We assume the network is strongly connected and channels are fault free.

We consider an execution of an algorithm as a sequence $a_1, \ldots, a_k, \ldots$ of *events* (cf. Lamport [Lam78]). The occurrence of these events is according to the program that the processes are running: an event takes place in a process only when the program of the process specifies the event and it is enabled. 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 same message is sent in a and 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 a particular execution as in [Lam78].

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

1. *$a = b$, or a and b occur in the same process and a happens earlier than b,*
2. *a is a send, and b the corresponding receive event, or*
3. *there is an event c such that $a \preceq c$ and $c \preceq b$.*

Because $\preceq$ is defined inductively we can prove statements of the form "$a \preceq b$ implies $P(a, b)$" by *induction over the definition of* $\preceq$. If (1) $a = b$ implies $P(a, b)$, and (a and b

occur in the same process **and** a happens earlier than b) implies $P(a,b)$; (2) a is a send, and b the corresponding receive event implies $P(a,b)$; and (3) $P(a,c)$ **and** $P(c,b)$ implies $P(a,b)$, then for all a and b, $a \preceq b$ implies $P(a,b)$.

Because a message can be received only after it is sent, for any execution $a_1, \dots, a_k$, $a_i \preceq a_j$ implies $i \leq j$. Thus $\preceq$ is a well-founded partial ordering. Because $\preceq$ is a well-founded partial ordering we can prove a property P of events using *induction over* $\preceq$. If for all a $(\forall b : b \preceq a \Rightarrow P(b))$ implies $P(a)$ then for all a $P(a)$. In words, if $P(b)$ for all b that precede a implies $P(a)$, then $P(a)$ holds for all a.

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.

Algorithms serve to compute some value or bring the network in a desired 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 depends on the problem under consideration. In section 2, 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. In a total algorithm it is required that a decision is preceded by the enroll events of all processes. Furthermore, a decision must be taken eventually 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,* $e_q \preceq d_p$. *An algorithm is total if all its executions are finite and each completed execution is total.*

This definition 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 when there is exactly one starter. An algorithm is *decentralized* if it works correct when any nonempty subset of the processes are starters.

2 Some Direct Applications of Total Algorithms

In this section we study the relation between total algorithms and some network control problems. We use the following well-known fact about asynchronous systems (which can be seen as the characterizing property of asynchronous systems) to show that algorithms for some problems are necessarily total. A proof is found in [Tel94, Sec. 2.3].

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

Informally, this theorem says that any reordering of the events in an execution that is consistent with the partial ordering $\preceq$, 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 [Seg83]. 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 fact that every process has accepted the message. This notification is what we referred to as a “decision” in the definition of total algorithms.

Theorem 2.2 *Every PIF algorithm is total.*

Proof. Assume P is a PIF algorithm. By definition it is required that in every execution at least one (accept) event takes place in every process and in at least one process a decision (notification of completion of the broadcast) takes place. Assume there is an execution of P where, for some process q, e_q does not precede a decision. From fact 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. Hence all executions of P must be total. □

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

Proof. Let A be a total algorithm. Processes that initiate 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 a decision is taken and for all q e_q precedes a 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 [Fin79]. 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”. (We drop the requirement of [Fin79] that all processes decide.)

Theorem 2.4 *Every Resynch algorithm is total.*

Proof. Assume R is a Resynch algorithm. By definition it is required that in every execution of R at least one event takes place in every process and in at least one process a decision takes place. Assume in some execution of R, for some process q, e_q does not precede a decision. From fact 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. It follows that all executions of R must be total. □

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 to *synch* upon first participation in A, i.e., in e_q. Each process p changes state to *normal* when it decides, i.e., in 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 $\wedge$. The Distributed Infimum computation problem asks for the following: suppose all processes p in $\mathbb{P}$ have an input $r_p \in X$. Compute the infimum $J = \inf\{r_p : p \in \mathbb{P}\}$. 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 $x \not\leq y$.

Theorem 2.6 *Every Infimum algorithm is total.*

Proof. Suppose I is an Infimum algorithm and S is an execution of I where no enrollment of q precedes 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 input r_q. Choose r_q such that $x \not\leq r_q$, we now have an execution in which p decides on a wrong result. □

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 leave i_p unchanged. If and when p decides in A, p outputs i_p as the result of the computation.

It must now be shown that this answer is correct. For an event a 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 the definition of $\preceq$ it is seen that $a \preceq b$ implies $i^{(b)} \leq i^{(a)}$. 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 $i^{(d)} = J$, and the constructed algorithm is correct. □

The problem of Distributed Infimum computation as described here is different from the problem of Distributed Infimum Approximation as described in [Tel86]. Here we consider fixed values r_p. In the Distributed Infimum Approximation problem changing values x_p are considered. The problem of Distributed Infimum Approximation is briefly addressed in section 5. The *Connectivity* problem, where processes are to compute the set $\mathbb{P}$, is a special case of Infimum Computation as $\mathbb{P} = \cup_{p\in\mathbb{P}}\{p\}$.

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 subclass of the can be proved, however. To this end it is necessary to make a (technical) restriction on the algorithms considered, namely that they are *comparison algorithms.* A comparison algorithm is an algorithm that allows comparison as the only operation on identities. For a more precise definition, see Gafni *et al.* [GLT$^+$85].

Theorem 2.10 *Comparison Election algorithms for rings, trees, or general networks of 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 different values.

For rings and trees similar constructions can be given. □

Many known Election algorithms are total, but non-total ones are also known, for example Peterson's algorithm for grid networks [Pet85] or Bar–Yehuda's alorithm for networks of known size [BYKWZ87]. See also section 4.4.

2.5 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. For the application of Traversal algorithms in [KKM90] it is required that the algorithm 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 centralized and sequential.

3 Some Examples of Total Algorithms

The total algorithms in this section appear in the literature but they are referred to as Distributed Infimum algorithm, PIF algorithm, etcetera, rather than as a "total algorithm". 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.

The algorithms include centralized as well as decentralized ones. 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 (RING algorithm) that uses N messages, which is (order) optimal, whereas for decentralized algorithms on a ring $O(N \log N)$ is optimal (Peterson's algorithm). For a general network (of N processes and E edges) there is a centralized algorithm using $2E$ messages (ECHO algorithm), which is (order) optimal, whereas for decentralized algorithms on general networks $O(N \log N + E)$ is optimal (GHS algorithm). 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. The TREE algorithm below is symmetric, but not decentralized.

In some algorithms all processes decide, in others only one or few. In the original statement of some problems, e.g., Resynchronization [Fin79], 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.

3.1 RING Algorithm

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 <> 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:

```
$\mathbf{S}_p$: { Spontaneous start, execute only once }
     *send* <> to successor

$\mathbf{R}_p$: { 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$_p$: { $Rec_p$ }
        begin $send$ <> to successor ; $Rec_p := false$ end
```

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 $g_i \preceq f_i$ if i is not l, and we have $f_i \preceq g_{i+1}$ because these are corresponding events. Only the starter decides, and we have $g_l \preceq d_l$ directly from the algorithm text. Thus, using the fact that there is only one starter, we find

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

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

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

and

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

but the execution is not necessarily total.

The RING algorithm 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.

3.2 TREE Algorithm

A total algorithm for a tree network. 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. Because leaves of the tree have only one link they must be starters (possibly except one). 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.

```
R_p: { A message <> from q arrives at p }
     begin receive <> ; Rec_p[q] := true end

S_p: { q ∈ Neigh_p is such that ∀r ∈ Neigh_p, r ≠ q : Rec_p[r] }
     begin send <> to q end

D_p: { for all q ∈ 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

$$T_{pq} = \{p\} \cup \bigcup_{r \in Neigh_p - \{q\}} T_{rp} \tag{1}$$

and

$$\mathbb{P} = \{p\} \cup \bigcup_{r \in Neigh_p} T_{rp}. \tag{2}$$

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

Proof. By induction on $\preceq$. Assume the lemma is true for all receive actions that precede g_{pq}. Let $s \in T_{pq}$. By 1, $s = p$ or $s \in T_{rp}$ for some $r \in Neigh_p$, $r \neq q$. We have $f_{pq} \preceq g_{pq}$ because these events correspond, $e_p \preceq f_{pq}$ because e_p precedes all events in p, so $e_p \preceq g_{pq}$ follows. By virtue of the algorithm $g_{rp} \preceq f_{pq}$ for all neighbors $r \neq q$ of p, and by the induction hypothesis $e_s \preceq g_{rp}$ for all s in T_{rp}. So $e_s \preceq g_{pq}$ follows. □

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

Proof. By 2, for all $s \in \mathbb{P}$, $s = p$ or $s \in T_{rp}$ for some $r \in Neigh_p$. We have $e_p \preceq d_p$ as above. If $s \in T_{rp}$, then we have $g_{rp} \preceq d_p$ by the algorithm, $e_s \preceq g_{rp}$ by the previous lemma, and $e_s \preceq d_p$ follows. □

In contrast with the RING algorithm, 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 events of the TREE algorithm 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 is ready to 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 (if there are at least 2 processes). 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 1 and 2, 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.

The TREE algorithm 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 $O(D)$ (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 (possibly except one) are starters. Hence not every nonempty subset of the processes suffices to start the algorithm, and the algorithm is not decentralized. We can transform the TREE algorithm to make it decentralized: starters flood a "wake-up" signal over the tree to trigger execution of the algorithm in 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 the TREE algorithm.

3.3 ECHO Algorithm

A centralized total algorithm for general bidirectional networks. This algorithm is usually referred to as Chang's Echo algorithm [Cha82]. The starter sends a message over all links. Followers store 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 the TREE algorithm. The initial value of the variable $father_p$ is *nil*. In the program fragment labeled with $\mathbf{S}_p$ 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 brevity. The same remarks apply to $\mathbf{R}_p$ below and later fragments. The program for the starter is:

```
$\mathbf{S}_p$: (* Spontaneous start, execute only once *)
     forall $r \in Neigh_p$ do $send$ <> to $r$
```

```
$\mathbf{R}_p$: { A message <> arrives from $q$ }
    begin receive <> ; $Rec_p[q] := true$ end

$\mathbf{D}_p$: { $\forall q \in Neigh_p : Rec_p[q]$ }
    decide
```

The program for a follower is:

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

$\mathbf{S}_p$: { $\forall q \in Neigh_p : Rec_p[q]$ }
    send <> to $father_p$
```

It turns out that the ECHO algorithm builds a spanning tree in the network and works as the TREE algorithm on this (rooted) tree. The following correctness proof is found in [Seg83]. Consider a (completed) execution S of the ECHO algorithm 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 $e_{father_p} \preceq e_p$ is easily derived from the algorithm), and (3) each non-starter has an outgoing edge (because, if a process p sends to its neighbor r, r eventually receives this message and then (i) r is starter, (ii) r assigns the value p to $father_r$, or (iii) $father_r$ 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, $e_q \preceq d_l$.*

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 $e_r \preceq g_{rp}$. As in the proof of theorem 3.2 the result follows. □

Theorem 3.6 *Assume that events of the ECHO algorithm that are enabled will eventually be executed. Then the starter eventually decides.*

Proof. We first show that until a decision is taken there is always a next event enabled to be executed. If there is a message underway this message will eventually be received. Assume that there is no message underway, it follows that each neighbor of a process that has sent has received a message. Because a follower sends to every neighbor as soon as it receives the first message, it follows that all processes have sent to every neighbor. Because no messages are underway all processes have received a message from every neighbor, hence the starter is ready to decide. Finally, remark that only finitely many messages are sent and received. $\square$

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 the RING algorithm, the execution is not necessarily total.

The ECHO algorithm 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 $2E$ (where E is the number of bidirectional edges), its time complexity is $O(D)$.

3.4 PHASE Algorithm

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 $i+1$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.

```
**R**$_p$: { A message <> arrives from $q$ }
      **begin** *receive* <> ;
            $RCount_p[q] := RCount_p[q] + 1$ **end**

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

**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 ith event in which p sends to q, and $g_{pq}^{(i)}$ be the ith event in which q receives from p. If a FIFO discipline on links is assumed these events correspond, so that trivially $f_{pq}^{(i)} \preceq g_{pq}^{(i)}$. We do not assume a FIFO discipline so that the following result becomes non-trivial.

Theorem 3.7 $f_{pq}^{(i)} \preceq g_{pq}^{(i)}$.

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

The lemma holds not only for the PHASE algorithm, but for any algorithm, even if 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 the PHASE algorithm.

Theorem 3.8 *For all $s \in \mathbb{P}, e_s \preceq d_p$.*

Proof. Let $p_0p_1 \dots p_l$, $l \leq D$, be a path in the network. By the previous theorem we have $f^{(i+1)}_{p_ip_{i+1}} \preceq g^{(i+1)}_{p_ip_{i+1}}$ for $i < l$ and by the algorithm we have $g^{(i+1)}_{p_ip_{i+1}} \preceq f^{(i+2)}_{p_{i+1}p_{i+2}}$ for $i < l-1$. Thus $e_{p_0} \preceq g^{(l)}_{p_{l-1}p_l}$. Because the network diameter is D, for every s and p there is a path $s = p_0 \dots p_l = p$ of length at most D. Thus $e_s \preceq g^{(l)}_{p_{l-1}p}$. By the algorithm $g^{(l)}_{p_{l-1}p} \preceq d_p$, so that the result follows. □

Theorem 3.9 *Assume that all events of the PHASE algorithm that are enabled will eventually be executed. Then all processes will decide.*

Proof. First we show that, until 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 $SCount$ variables 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 there is always an event enabled until all processes have decided. Furthermore, only a finite number of events can take place, namely DE send, DE receive, and N decide events. It follows that all processes will decide. □

The PHASE algorithm is studied further in [Tel91, Sec 4.2.3]. The algorithm 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 DE, the time complexity of the PHASE algorithm is D.

The algorithm can considerably be simplified if it is used on a complete network ($D = 1$). All a process must do is sent a message to every other process, receive a message from every other process (i.e., $N-1$ messages altogether), and decide. Formally it is expressed as follows (the initial value of $RecCount_p$ is 0):

```
S_p: (* Execute once *)
     forall $q \in \mathbb{P} - \{p\}$ do send <> to q

R_p: { A message <> arrives }
     begin receive <> ; $RecCount_p := RecCount_p + 1$ end

D_p: { $RecCount_p = N - 1$ }
     decide
```

3.5 GOSSIP Algorithm

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 $OK_p = \varnothing$. 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 for each process p, In_p be the set of its in-neighbors, Out_p be the set of its out-neighbors, and p has a boolean variable $Rec_p[q]$ for each $q \in In_p$. The value of $Rec_p[q]$ is initially $false$ and indicates whether a message from q has been received. In the following algorithm, the **S** and **R** fragments may be executed more than once.

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

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

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

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 3.10 *If $e_q \preceq b$ 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, $c \preceq b$ implies $HO^{(c)} \subseteq HO^{(b)}$. By the algorithm $q \in HO^{(e_q)}$, so the result follows. □

Lemma 3.11 *If $q \in OK^{(b)}$ then for all $r \in In_q, e_r \preceq b$.*

Proof. First we prove by induction on $\preceq$ that $q \in OK^{(b)}$ implies $a_q \preceq b$. Let p be the process where b occurs and let b' be the first event in p such that $q \in OK^{(b')}$. Now $b' \preceq b$ and before b' $q \notin OK_p$, because initially OK_p was empty. b' is either a receive event or a_p, for send and decide events leave OK_p unchanged. If $b' = a_p$ then $p = q$ and $a_q \preceq b$ follows. If b' is a receive event, q was contained in the OK set received. Use (the induction hypothesis on the corresponding send event c) $a_q \preceq c$ and conclude $a_q \preceq b$. Thus $q \in OK^{(b)}$ implies $a_q \preceq b$.

From the algorithm it follows that for all $r \in In_q$, $e_r \preceq a_q$, from which the result follows immediately. □

Theorem 3.12 *For all r, $e_r \preceq d_p$.*

Proof. Trivially $e_p \preceq d_p$. If q is a process such that $e_q \preceq d_p$, then by 3.10 $q \in HO^{(d_p)}$, so by $HO^{(d_p)} = OK^{(d_p)}$ we have $q \in OK^{(d_p)}$, and by 3.11 we find that for all $r \in In_q$, $e_r \preceq d_p$. The result follows from the strong connectivity of the network. □

The message complexity of the GOSSIP algorithm as given is unbounded, because a send event can in principle be repeated infinitely often. We 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 (where E is the number of unidirectional edges). Under this restricted sending policy we can prove:

Theorem 3.13 *Assume all events of the GOSSIP algorithm that are enabled will eventually be executed. Then all processes will decide.*

Proof. We first show that, until 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 an execution does not come to a halt 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. □

The GOSSIP algorithm is in fact Finn's Resynch algorithm [Fin79]. 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 the GOSSIP 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 mechanism was described separately by Segall [Seg83].

A consequence of the material in this section is that the Resynch problem [Fin79] can be solved by an algorithm using fewer messages. In Finn's work bidirectional channels are assumed, so the GOSSIP algorithm can be replaced by e.g. the GHS algorithm. For the resynchronization of unidirectional networks the algorithm of Gafni and Afek [GA84] can be used instead. An advantage of the GOSSIP algorithm over the others is its low time complexity.

The GOSSIP algorithm 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 the TREE, ECHO, PHASE, and GOSSIP algorithms. The TREE algorithm assumes a tree topology, the ECHO algorithm assumes there is exactly one starter, the PHASE algorithm assumes the network diameter to be known, and the GOSSIP algorithm assumes processes have identities. It can be shown that at least one of these assumptions is necessary: there exists no decentralized total algorithm for anonymous, general networks with no bound on the diameter.

3.6 Other Total Algorithms

We briefly describe a few other total algorithms, without correctness proof.

The REPLY algorithm. A centralized total algorithm for complete networks. The starter sends a message to every other process. A follower acknowledges the receipt of a message. The starter decides when it has received enough acknowledgements.

The GHS algorithm. 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 adaptation by Gafni and Afek to directed networks [GA84].

Many algorithms have been given for Election on (unidirectional) rings. Their complexity is typically $O(N \log N)$. Peterson's algorithm [Pet82] 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.

Distributed Depth First Search (DDFS) [Tel94, Sec. 6.4]. The classical DDFS algorithm [Che83] is a centralized algorithm for general bidirectional networks. It is sequential, i.e., DDFS is a Traversal algorithm in the sense of [KKM90]. 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] or [Tel94]), but these variants are no longer sequential. A version for directed networks exists [GA84].

4 Mutual Exclusion and Election

Mutual Exclusion is a fundamental problem in the design and implementation of parallel and distributed systems. It must be solved if the system contains resources, such as printers, shared variables, or telephone connections, that can be used by several processes, but by at most one at a time. To model such resources, we assume that the code of processes may contain so-called *critical sections*. At any time at most one process may be executing in its critical section. This means that before entering its critical section, a process must go through a protocol to ensure that no other process is in its critical section or will enter it while the critical section is executed. If necessary the entering of the critical section is deferred.

The protocol that is used for this task is called a *Mutual Exclusion* protocol. This protocol must satisfy the following criteria.

1. *Mutual Exclusion* (or safety): at most one process is in its critical section at a time.

2. *Starvation-freeness* (or liveness): a process that starts the protocol to enter its critical section will in finite time be enabled to enter its critical section.

To satisfy starvation-freeness, the assumption must be made that if a process is in its critical section, it will leave it in finite time.

In this section we will show that a Mutual Exclusion protocol can be obtained by a superimposition on a suitable total algorithm. In fact, the superimposition is a generalization of the Mutual Exclusion algorithm of Ricart and Agrawala [RA81]. Their algorithm is obtained by applying the superimposition to the REPLY algorithm of section 3. Application of the superimposition to the ECHO algorithm yields an algorithm, very similar to that of Hélary *et al.* [HPR88].

The idea of the algorithm is very simple and ressembles the bakery principle. Each request to enter the critical section is assigned a unique identification label v. Before entering the critical section, the requesting process starts an execution $A^{(v)}$ of the total algorithm to certify that its label is the smallest in the network. A process, executing or requesting to execute a critical section with a label $w < v$ defers its participation in $A^{(v)}$. A process that has participated in $A^{(v)}$ will thereafter never issue a request with an identification $w \leq v$. A process enters its critical section labeled v if the corresponding $A^{(v)}$ has lead to a decision. Upon leaving the critical section, deferred executions are resumed.

The Mutual Exclusion algorithm does not rely on a special message, a *token*, possession of which is necessary to enter the critical section. Thus it is not necessary to select one process as the original possessor of the token, and there is no danger of token loss when a process crashes. The algorithm is symmetric, but to break ties it is required that processes have a unique identity.

We see that in the Mutual Exclusion algorithm several executions of the underlying total algorithm take place. A similar approach has been used for the construction of Election algorithms. Here the aim is to construct a *decentralized* Election algorithm, based upon *centralized* total algorithms, such as the RING, ECHO, or REPLY algorithm of section 3. The idea is that starters of the Election algorithm start a copy of the underlying centralized total algorithm, and these copies compete until finally one copy results in a decision. We mention some of these constructions because there turns out to be a connection between these constructions and the construction of Mutual Exclusion algorithms.

In section 4.1 describe how a class of algorithms for the Mutual Exclusion problem can be constructed, based on suitable total algorithms. The algorithms are easy to understand, to implement, and to prove correct. Because they do not rely on the possession of a unique message, a token, to obtain access to the critical section, they can easily be made fault-tolerant. In section 4.2 we give its correctness proof. In section 4.3 we deal with the construction of Election algorithms. It turns out that totality is too strong a requirement for the underlying algorithm in these constructions. Therefore in section 4.4 we define the weaker notion of *dominating* algorithms. We conjecture that dominating algorithms also allow elegant constructions yielding algorithms for Mutual Exclusion and Election, but the details of the construction are left open for further research. A thorough study of dominating algorithms was not the aim of this section. Investigations in this direction may lead to the development of new and efficient algorithms for these problems.

4.1 Construction of Mutual Exclusion Algorithms

In the following A stands for a total algorithm. A must satisfy the following requirement: *If there is one starter (in a particular execution) then this process eventually decides.* This is not equivalent to A being centralized, as defined in section 3. Executions of A are tagged with an execution identification v. This means that messages of the execution are augmented with v, and separate state variables exist in each process for each execution. Each execution has only one starter, but several executions can take place simultaneously. Executions do not influence each other (except via the superimposition we will describe). We refer to the execution tagged with v as $A^{(v)}$.

A tag v consists of two components $v.1$ and $v.2$, where $v.1$ is a sequence number, unique for a process, and $v.2$ is the identity of the process that started the execution. We order the tags totally by a lexicographical order, i.e., $v \leq w$ means $v.1 < w.1$ **or** $(v.1 = w.1$ **and** $v.2 \leq w.2)$.

The algorithm consists of the following procedures. The procedure *RequestCS* is executed when a process wants to enter its critical section. A unique sequence number is chosen and an execution of A started. The procedure *Decide*(v) is called when algorithm $A^{(v)}$ decides. The critical section is entered if the corresponding request was issued in this process. The procedure *LeaveCS* is executed when the process leaves its critical section. Deferred executions of A are resumed.

Whenever an event of an execution $A^{(w)}$ of A is enabled, w is compared to the identity v of a possibly pending request of the process itself. If $v \leq w$ the execution of the event is deferred. If it is a receive event, the message is stored in a queue *MesQ*. To simplify the description of the algorithm we assume that a process is involved in at most one request or critical section at a time. That is, when a process has a request pending or is in its critical section, no new requests are generated in the process.

Variables of a process are the following. A boolean $Request_p$ indicates whether or not p has a request outstanding. If this is the case, $ReqID_p$ contains the label of the outstanding request. $SeqNR_p$ contains the highest sequence number (i.e., first component of any tag) p has seen so far. $MesQ_p$ contains messages of A whose receipt has been deferred.

```
$RequestCS_p$:  (* $p$ wants to enter its critical section *)
                begin $SeqNR_p := SeqNR_p + 1$ ;
                      $ReqID_p := (SeqNR_p, p)$ ;
                      $Request_p := true$ ;
                      START $A^{(ReqID_p)}$ ; (* start a new execution of $A$ *)
                end

$Decide_p^{(v)}$:   (* $p$ decides in execution $A^{(v)}$ *)
                begin if $v.2 = p$ then (* this was "my" execution *)
                        enter critical section
                end

$LeaveCS_p$:    (* $p$ leaves critical section *)
                begin $Request_p := false$ ;
                      Execute all deferred events of $A$, using $MesQ_p$
                end

{ A message $M$ of $A^{(v)}$ arrives }
                begin receive $M$ ;
                      $SeqNR_p := \max(SeqNR_p, v.1)$ ;
                if $Request_p$ and $ReqID_p < v$
                   then append $M$ to $MesQ_p$
                   else execute receive event of $A^{(v)}$
            end

{ A send or internal event $e$ of $A^{(v)}$ is enabled }
            begin if $Request_p$ and $ReqID_p < v$
                   then defer $e$
                   else execute $e$
            end
```

4.2 Correctness of the Algorithm

Each request to enter the critical section is assigned a request label, consisting of a sequence number and a process identity. We use the label to identify the request itself, the resulting execution of A, and the corresponding execution of the critical section as well.

Lemma 4.1 *Request labels are unique, i.e., different requests have different labels.*

Proof. As the second component of the request label is the processor identity, requests of different processes have different labels. As $SeqNR_p$ is incremented to compute the request label (see procedure $RequestCS_p$), requests of the same process have increasing, and thus different, labels. □

Theorem 4.2 *(Mutual Exclusion) At most one process is executing in its critical section at a time.*

Proof. From the algorithm it follows that after $e_p^{(v)}$, p will not issue a request with label $w < v$ or execute a critical section with label $w < v$. ($e_p^{(v)}$ is the enroll event of p in $A^{(v)}$.) Entering of a critical section with request label v is preceded by a decision in $A^{(v)}$, and thus, by totality of A, by $e_p^{(v)}$, for all p. Thus, by the previous remark, a critical section with request label v is not executed concurrently with a critical section with a label $w < v$. Because labels are unique, Mutual Exclusion follows. □

Theorem 4.3 *(Starvation freeness) A process, requesting to enter its critical section, will be enabled to do so within finite time.*

Proof. Assume there are pending requests to enter the critical section. Consider the request with lowest label v. $A^{(v)}$ is deferred only by nodes with a pending request or an executing critical section with a label smaller than v. Critical sections are eventually left and undeferred executions of A eventually decide. It follows that within finite time a new request (with label $< v$) is generated or request v is granted. The corresponding critical section will be entered and, by assumption, be left in finite time. So, a next critical section will always be entered or a new request with a smaller label will be generated in finite time. Critical sections are always executed in strictly increasing order of request labels. For each request, there is only a finite number of possible requests with a smaller label. Thus all requests will lead, in finite time, to the entering of the corresponding critical section. □

4.3 Construction of Election Algorithms

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 Election algorithms from centralized total algorithms: 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 an Election 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 starts a copy of C (unless it did so earlier), but does not participate in C_q, i.e., it does not execute any event of the execution C_q. If $q \geq m_p$, p updates m_p and participates in C_q. If a decision is taken in C_q, this is regarded as a decision in E, and q is elected.

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 Extinction construction can be improved as follows. When a process p has not started C_p (yet) and receives a message it participates in the corresponding execution and never starts C_p 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] is obtained by applying Extinction to the RING algorithm. The algorithm by Hirschberg [Hir80] is obtained by applying Extinction to the ECHO algorithm.

Construction 2: (Korach *et al.*, [KKM90]) Starting from a traversal algorithm with message complexity $f(N)$, Korach *et al.* construct an Election algorithm with message complexity $O((f(N) + N)\log s)$.

Construction 3: (Attiya, [Att87]) Starting from a traversal algorithm with message complexity $f(N)$, Attiya constructs an Election algorithm for bidirectional 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 while construction 1 can use a centralized, but not sequential, total algorithm.

4.4 Dominating Algorithms

For the purpose of Election and Mutual Exclusion the requirement that enrollment of every process precedes a decision is too strong. Already in section 2.4 we saw that not all Election algorithms are total. For these problems it suffices that no two processes can decide independently. We restrict ourselves now to algorithms with the mentioned property that if the algorithm is started by a single process p, p will eventually decide. Let B be such an algorithm. We refer to an execution in which p is the starter as $B^{(p)}$.

Events in $B^{(p)}$ are denoted by $e_q^{(p)}$, $d^{(p)}$, etc.

Definition 4.4 *B is* dominating *if for any two executions $B^{(p)}$ and $B^{(q)}$ there is a process r such that $e_r^{(p)} \preceq d^{(p)}$ and $e_r^{(q)} \preceq d^{(q)}$.*

It appears that a dominating algorithm B can be used to construct an Election algorithm. To start an Election, a process p starts $B^{(p)}$. When $B^{(p)}$ decides p declares itself leader. Because B is dominating, it is impossible that two processes p and q both declare themselve leader if any process r refuses to take part in both $B^{(p)}$ and $B^{(q)}$. Conflicts of this type must be solved, but we did not yet work out a construction to do this.

It appears that a dominating algorithm B can be used to construct a Mutual Exclusion algorithm. To request the critical section, a process chooses a unique request label $v = (SeqNR, p)$ as in our construction, and starts $B^{(v)}$. Upon decision of $B^{(v)}$ p enters the critical section. An execution $B^{(v)}$ is now deferred not only by processes with a pending request with smaller label, but also by processes that are "blocked" by having participated in an algorithm $B^{(w)}$, $w < v$. Upon leaving the critical section p again executes $B^{(p)}$, now to release the blocked processes. Note that for this purpose B must be deterministic in the sense that each execution of $B^{(p)}$ involves the same set of processes. Additional control is necessary to prevent deadlock in this scheme.

In both cases the constructions are more complex than similar constructions from total algorithms. Of course the concept of dominating algorithms is useful only if dominating algorithms are not the same as total algorithms. This is the case indeed. On the one hand, total algorithms are dominating: in a total algorithm A, for *each* process r, $e_r^{(p)} \preceq d^{(p)}$ and $e_r^{(q)} \preceq d^{(q)}$. On the other hand, not all dominating algorithms are total. With an argument similar to the one used in theorem 2.10 it is easily seen that a dominating algorithm for rings, trees, or general networks of unknown size is necessarily total. In such networks an unvisited node may "hide" a large subnetwork where an identical execution can take place, thus violating the assumptions of the algorithm. There exist some non-trivial dominating algorithms, which have a message complexity or other properties that can not be attained by total algorithms.

Algorithm 1: For networks with known size N. The starter floods a request through the network and decides when it has received replies from $\lfloor N/2 \rfloor$ processes. The message complexity is of the same order as of the REPLY algorithm, but this algorithm is highly fault-tolerant. It is the basis of an Election algorithm by Kutten [BYKWZ87].

Algorithm 2: For networks with a designated process L. L replies to received messages by sending a message back. The algorithm for a starter p consists of sending a message to L, and awaiting the reply. Algorithm 2 implements centralized control of the network.

Algorithm 3: For grid or torus networks. The starter sends tokens along its column and row, and receives them back. This algorithm is dominating because each row crosses each column. A similar approach can be used in networks whose topology is a projective plane. The message complexity can be $O(\sqrt{N})$. This algorithm underlies the Election algorithm of Peterson [Pet85] (torus) and the Mutual Exclusion algorithm of Maekawa [Mae85] (projective plane).

5 Distributed Infimum Approximation

Distributed Infimum Approximation was defined in [Tel86] as an abstraction of several control problems in distributed systems, including Termination detection and evaluation of Global Virtual Time. Several constructions of algorithms for the problem were given in [Tel86]. This section contains a brief overview of the material contained in that paper.

5.1 Definition of the Problem

Each process p is equipped with a variable x_p with values in a partially ordered domain X. A so–called *basic computation* manipulates these local variables according to the following rules:

send Whenever p sends a message it includes the current value of x_p in the message and x_p remains unchanged.

receive Whenever p receives a message with a value x contained in it, p assigns to x_p the infimum of x_p and x: $x_p := x_p \wedge x$.

internal If an internal event in p changes the value of x_p, it only *increases* this value: x_p is assigned a value $x' \geq x_p$.

A function on global system states is defined as follows ($x_p^{(S)}$ denotes the value of variable x_p in state S):

$$F(S) = \inf(\{x_p^{(S)} : p \in \mathbb{P}\} \cup \{x : x \text{ is contained in any message in } S\}).$$

From the rules of the computation, given above, it follows that F is a monotone function, that is, if S_2 is a later system state than S_1, $F(S_1) \leq F(S_2)$.

The Distributed Infimum Approximation problem asks to superimpose an algorithm on this basic computation such that an *approximation* f of F is maintained, satisfying

1. (*Safety*) $f \leq F$ at any time, and
2. (*Progress*) if $F \geq k$ (for some $k \in X$) then within finite time $f \geq k$ also.

To show that the Termination Detection problem is an instance of Distributed Infimum Approximation, take $X = \{active, passive\}$ ordered such that $active < passive$ and observe that the computation is terminated if and only if $F = passive$.

5.2 Construction of Distributed Infimum Approximation Algorithms

In this paper we only give a construction for a special case of the problem, namely, where communication in the basic computation is instantaneous. In this case

$$F(S) = \inf(\{x_p^{(S)} : p \in \mathbb{P}\}).$$

To compute successive approximations f the execution of a total algorithm, say A, is repeated many times. Repeated execution is also required in other applications, such as the algorithm for distributed selection of Santoro *et al.* [SSS88]. Iteration schemes based on repeated executions of total algorithms were studied also by Hélary and Raynal [HR88]. In all these applications it is required that subsequent executions of the algorithm are *disjoint*. Let A be a total algorithm which is executed repeatedly, and let $e_p^{(i)}$ denote p's enroll event in the i^{th} execution of A.

Definition 5.1 *A series of executions of A is called* disjoint *if for all p, q, and i: $e_p^{(i)} \preceq e_q^{(i+1)}$.*

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 $e_p \preceq e_q$. This fact, together with the rule, implies indeed that the subsequent executions are disjoint. The algorithm A must be such that if the deciders are starters in a next round, the next execution is guaranteed to be total also. This condition is satisfied if A is decentralized (PHASE, GOSSIP), but also if there is one starter and one decider (RING, ECHO, REPLY).

By a superimposition on A as in section 2.3 it is arranged that $A_{(i)}$ computes the infimum of values $r_p^{(i)}$, provided by the processes in event $e_p^{(i)}$. The value $r_p^{(i)}$ is computed by process p by observation of the basic computation in process p between $e_p^{(i-1)}$ and $e_p^{(i)}$. (For convenience, denote by $e_p^{(0)}$ the start of the basic computation in p.) Of course the local observations must be such that the outcomes of the series of executions of A satisfy the requirements for f. In the case we consider here (instantaneous communication of the basic computation) the value $r_p^{(i)}$ is simply the infimum over all values of x_p between $e_p^{(i-1)}$ and $e_p^{(i)}$. Assume that between enroll events this value is maintained in the variable r_p. Then the construction of a DIA algorithm can be summarized in the following rules:

R1 Initially $r_p = x_p$. Whenever x_p changes value, execute $r_p := r_p \wedge x_p$.

R2 Modify A as to compute infimums and such that a process that decides starts a next execution.

R3 Upon each enroll event of an execution of A in p, p uses r_p as input for A and sets $r_p := x_p$.

R4 Initially $f = \bot$. The outcome of an execution is assigned to f.

5.3 Correctness Proof

The proof of safety of the described algorithm is based on the fact that there is a system state S_1 that lies between two conseqtive "waves" of enroll events.

Theorem 5.2 *$f \leq F$.*

Proof. Let S_2 be any system state after the assignment of the outcome of $A^{(i)}$ to f. By the disjointness of the series of executions of A there is a system state S_1 such that, for each process p, $e_p^{(i-1)}$ takes place *before* S_1 and $e_p^{(i)}$ takes place *after* S_1. So by virtue of the observation mechanism, $r_p^{(i)} \leq x_p^{(S_1)}$ and thus $\inf_{p \in \mathbb{P}} r_p^{(i)} \leq \inf_{p \in \mathbb{P}} x_p^{(S_1)} = F(S_1)$. The left hand side of this inequation is the value assigned to f, while the right hand side is smaller that $F(S_2)$ by the monotonicity of F. □

Theorem 5.3 *If $F \geq k$ then within finite time $f \geq k$.*

Proof. Let i be the number of the first execution of A that starts after $F \geq k$. Now for all p $r_p^{(i+1)} \geq k$ so after completion of the $(i+1)^{\text{th}}$ execution $f \geq k$. □

Examples of these constructions are found in [Tel86] and [Tel91].

6 Conclusions

Total algorithms are an important building block in the design of distributed algorithms for a wide range of network problems. Algorithms for Propagation of Information with Feedback, for Resynchronization, for Infimum Computation, and for Connectivity can be constructed by adding information to the messages of any total algorithm. Algorithms for Mutual Exclusion and Election can be obtained by a more complex superimposition on a total algorithm. Algorithms for Distributed Infimum Approximation can also be obtained by a construction based on total algorithms.

The notion of total algorithms and the results in section 2 establish an equivalence between communication (Propagation of Information with Feedback), computation (Distributed Infimum), and synchronization (Resynch). This equivalence follows from fact 2.1 and holds in asynchronous networks only. In fact, in synchronous systems it is possible to achieve synchronization using clocks instead of communication.

The concept of total algorithms allows us to abstract away from the network topology when designing an algorithm. The underlying total algorithm is designed separately. Until now, independency of the topology was always achieved by using an algorithm for general networks [SE85, Fin79, HPR88]. Our approach combines separation of concerns with the possibility to use more efficient total algorithms whenever a particular topology allows to do so.

Reasoning in this paper was based on considering complete executions and a partial order $\preceq$ on the events. This can be seen as an alternative for reasoning about system states only, as advocated for example in [CM88]. The partial order allows us to express relations necessary for example in the proof of theorem 2.7. Thus, although it is generally felt to be less error–prone to reason about system states only, it is doubtfull whether all our results could be obtained by it.

Why are total algorithms so important? An execution of a total algorithm involves all processes, and can spread information to all processes as well as collect information from all processes. In fact, to execute a total algorithm the processes of a distributed system cooperate as one. The main burdens of distributed programming, namely lack of global view and lack of global control, are thus lifted from the fragile shoulders of the humble programmer.

References

[Att87] Attiya, H. Constructing efficient election algorithms from efficient traversal algorithms. In Proc. *2nd Int. Workshop on Distributed Algorithms* (Amsterdam, 1987), J. van Leeuwen (ed.), vol. 312 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 377–344.

[Awe85] Awerbuch, B. A new distributed depth first search algorithm. *Inf. Proc. Lett.* **20** (1985), 147–150.

[BYKWZ87] Bar-Yehuda, R., Kutten, S., Wolfstahl, Y., and Zaks, S. Making distributed spanning tree algorithms fault-resilient. In Proc. *Symp. on Theoretical Aspects of Computer Science* (1987), F. J. Brandenburg et al. (eds.), vol. 247 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 432–444.

[CBT94] Charron-Bost, B. and Tel, G. Approximation d'une borne inférieure répartie. Rapport de Recherche LIX/RR/94/06, Ecole Polytechnique, 1994. Submitted to RAIRO.

[Cha82] Chang, E. J.-H. Echo algorithms: Depth parallel operations on general graphs. *IEEE Trans. Softw. Eng.* **SE–8** (1982), 391–401.

[Che83] Cheung, T.-Y. Graph traversal techniques and the maximum flow problem in distributed computation. *IEEE Trans. Softw. Eng.* **SE–9** (1983), 504–512.

[CM88] Chandy, K. M. and Misra, J. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988 (516 pp.).

[CR79] Chang, E. J.-H. and Roberts, R. An improved algorithm for decentralized extrema finding in circular arrangements of processes. *Commun. ACM* **22** (1979), 281–283.

[Fin79] Finn, S. G. Resynch procedures and a fail-safe network protocol. *IEEE Trans. Commun.* **COM–27** (1979), 840–845.

[GA84] Gafni, E. and Afek, Y. Election and traversal in unidirectional networks. In Proc. *Symp. on Principles of Distributed Computing* (1984), pp. 190–198.

[Gaf86] Gafni, E. Perspectives on distributed network protocols: A case for building blocks. In Proc. *MILCOM* (Monterey, California, 1986), pp. 1–5.

[GHS83] Gallager, R. G., Humblet, P. A., and Spira, P. M. A distributed algorithm for minimum weight spanning trees. *ACM Trans. Program. Lang. Syst.* **5** (1983), 67–77.

[GLT$^+$85] Gafni, E., Loui, M. C., Tiwari, P., West, D. B., and Zaks, S. Lower bounds on common knowledge in distributed systems. In Proc. *Distributed Algorithms on Graphs* (1985), E. Gafni and N. Santoro (eds.), Carleton University Press, pp. 49–67.

[Hir80] Hirschberg, D. S. Election processes in distributed systems. Tech. rep., Dept Computer Science, Rice University, Houston, Texas, 1980.

[HPR88] Hélary, J.-M., Plouzeau, N., and Raynal, M. A distributed algorithm for mutual exclusion in an arbitrary network. *Computer J.* **31** (1988), 289–295.

[HR88] Hélary, J.-M. and Raynal, M. Un schéma (abstrait) d'itération répartie. Rapport de Recherche 417, IRISA, Rennes, 1988.

[KKM90] Korach, E., Kutten, S., and Moran, S. A modular technique for the design of efficient leader finding algorithms. *ACM Trans. Program. Lang. Syst.* **12** (1990), 84–101.

[Lam78] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* **21** (1978), 558–564.

[Mae85] Maekawa, M. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Trans. Comput. Syst.* **3** (1985), 145–159.

[Pet82] Peterson, G. L. An $O(n \log n)$ unidirectional algorithm for the circular extrema problem. *ACM Trans. Program. Lang. Syst.* **4** (1982), 758–762.

[Pet85] Peterson, G. L. Efficient algorithms for elections in meshes and complete networks. Tech. Rep. TR 140, Dept Computer Science, University of Rochester, Rochester NY 14627, 1985.

[RA81] Ricart, G. and Agrawala, A. K. An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM* **24**, 1 (1981), 9–17.

[SE85] Skyum, S. and Erikson, O. Symmetric distributed termination. In *The Book of L*, G. Rozenberg and A. Salomaa (eds.), Springer-Verlag, 1985.

[Seg83] Segall, A. Distributed network protocols. *IEEE Trans. Inform. Theory* **IT–29** (1983), 23–35.

[SSS88] Santoro, N., Scheutzow, M., and Sidney, J. B. On the expected complexity of distributed selection. *J. Parallel and Distributed Computing* **5** (1988), 194–203.

[Tel86] Tel, G. Distributed infimum approximation. Tech. Rep. RUU–CS–86–12, Dept Computer Science, Utrecht University, The Netherlands, 1986.

[Tel91] Tel, G. *Topics in Distributed Algorithms*, vol. 1 of *Cambridge Int. Series on Parallel Computation.* Cambridge University Press, 1991 (240 pp.).

[Tel94] Tel, G. *Introduction to Distributed Algorithms.* Cambridge University Press, 1994.

Contents

1 Introduction **1**
1.1 Definition of Total Algorithms . . . 2

2 Some Direct Applications of Total Algorithms **3**
2.1 Propagation of Information with Feedback . . . 3
2.2 Resynchronization . . . 4
2.3 Distributed Infimum Computation . . . 5
2.4 Election . . . 5
2.5 Traversal . . . 6

3 Some Examples of Total Algorithms **7**
3.1 RING Algorithm . . . 7
3.2 TREE Algorithm . . . 8
3.3 ECHO Algorithm . . . 10
3.4 PHASE Algorithm . . . 12
3.5 GOSSIP Algorithm . . . 13
3.6 Other Total Algorithms . . . 16

4 Mutual Exclusion and Election **16**
4.1 Construction of Mutual Exclusion Algorithms . . . 17
4.2 Correctness of the Algorithm . . . 19
4.3 Construction of Election Algorithms . . . 20
4.4 Dominating Algorithms . . . 20

5 Distributed Infimum Approximation **22**
5.1 Definition of the Problem . . . 22
5.2 Construction of Distributed Infimum Approximation Algorithms . . . 22
5.3 Correctness Proof . . . 23

6 Conclusions **24**