Time-Driven Algorithms for Distributed Control

Paul M.B. Vitányi

C.W.I., Kruislaan 413, 1098 SJ Amsterdam

Distributed algorithms are investigated for clock synchronization, spanning tree construction and leader-finding in large store-and-forward networks of processors communicating by message passing. In the synchronization algorithm the clocks are allowed to drift in both value and speed; the message delivery delay is unknown and may change with time. The algorithm for distributed elections and distributed spanning tree construction uses time, yet is logically time independent. Using time, we obtain better performance in terms of message-passes and passed bits than is possible otherwise, and better performance than by any other known algorithm. The algorithm works correctly for any network topology, under any asynchronicity in the network, and assumes no global knowledge about the network.

1980 Mathematics Subject Classification: 68C05, 68C25, 68A05, 68B20, 94C99
CR Categories: C.2, D.4, F.2.2, G.2.2.
Keywords & Phrases: distributed clock synchronization, distributed spanning tree, distributed elections, algorithms using time, time-independent correctness and termination, robustness, accelerated efficiency by improved synchronicity, distributed control, computer networks, network topology.
Note: this paper is meant for publication elsewhere

1. Introduction

As large multiprocessor systems communicating by message passing start to be actually constructed (we give some examples in a later section), and on a geographically grander scale very large computer networks, synchronization problems connected with the operation of such complexes are bound to become acute. We exhibit clock synchronization algorithms, for synchronizing the local clocks of the processors, using less assumptions on clock drift and communication delay than has previously been done.
Another problem which gets crucial for very large computer complexes is the number of message passes. Without efficient congestion control the system will be swamped by communication messages effectively blocking throughput. The most efficient congestion control consists in designing message-thrifty algorithms. We exhibit an algorithm for distributed leader-finding and distributed spanning tree construction which works for any

Report CS-R8510
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

network, under any asynchronous behavior, with only local information, and is more thrifty in message passes and passed bits than any other known algorithm. In fact, the algorithm performs within a (small) constant multiplicative factor of the lower bounds.

The networks we consider are point-to-point (store-and-forward) communication networks described by an undirected communication graph $G=(V,E)$, with the set of nodes V representing the processors of the network, and the set of links E representing bidirectional noninterfering communication channels between them. No common memory is shared by the node-processors. Each node processes messages received from its neighbors, performs local computations on messages and sends messages to neighbors. All these actions take a finite time. All messages have a finite length according to the finite amount of information they carry. Each message sen$\overset{t}{\cancel{d}}$ by a node to its neighbor arrives there in a finite time. Messages over a given link arrive in the order in which they are send, and are stored in a receiving queue of unbounded length at the destination. We may think of messages as being chopped into fixed-length packets at the sender which are reassembled by the receiver. A *message pass* consists of the sending of a message from one node to one of its direct neighbors.

Synchronicity. Problems resulting from lack of synchronization are dealt with using logical time [15] or by constructing algorithms which can deal with unlimited asynchronism. Such algorithms can surely deal with any environment in which there is knowledge about processor speed and message delivery time. Unlimited asynchronous models have been thoroughly investigated, as have purely synchronous models. Physical systems are usually somewhere in between: they are neither purely synchronous nor unlimited asynchronous. It therefore is an interesting exercise to develop algorithms which do not use knowledge about the relative progress of time in the system, yet perform superior under realistic conditions about time. The usual logically time-independent algorithms do not assume anything about the rate at which time flows in different locations. This is unnecessa$\overset{il}{ry}$ harsh with respect to many problems arising in the actual world. The algorithm for approximate *clock synchronization* below sends three messages in a Z-leg between a pair of adjacent neighbors and computes the relative local times and timerates from the collected information. The reason why this algorithm is satisfactory is because whatever happens in the system happens with a certain smoothness. So also clock drift [16]. Abrupt changes are rare in nature. The distributed elections and spanning tree constructions presented below are other examples of which the superior performance relies on well-behaved nature. Unlike the mentioned synchronization, however, these algorithms are robust. They remain correct and terminate under any behavior of time in the system. Using time, the algorithms are yet logically time-independent; only their efficiency depends on the behavior of time. With more synchronous well-behaved time in the system the performance of the algorithms improves ever faster. If the asynchronicity of the system is known then the algorithm can be made to perform as well as in the synchronous case. Under practical assumptions about clock speeds these algorithms use less message passes than is possible by any other known method. This is an initial investigation into the concept of *time-driven* algorithms for distributed control in synchronous or asynchronous systems. The algorithms for distributed spanning tree and distributed elections have a good performance on paper; yet it is unlikely that they are, in their present form, useful in a real environment. This question is addressed in more detail later. The primary goal of the present investigation is to demonstrate the existence of competitive time-driven algorithms with the desirable properties as mentioned. We expect that genuinely more efficient algorithms, than the unlimited asynchronous ones, exist in between the pure synchronous and unlimited asynchronous ones.

Clock Synchronization. For synchronization purposes in multiprocessor systems the individual processors often need to maintain clocks that are synchronized with one another. Physical clocks cannot keep perfect time; therefore clocks can drift with respect to one another both in the time they indicate and the rate at which time flows. Clock synchronization algorithms have been studied extensively [15,1], Recent work [16] contains solutions for difficult problems of distributed synchronization in the presence of faults. These algorithms must deal with the malicious presence of "two-faced" clocks which present one time to one observer and another time to another observer simultaneously. See also [6,7]. Although we investigate milder irregularities of clocks than [16], the assumptions we make about the clocks are weaker.

Distributed Spanning Tree. Consider a connected undirected graph G with N nodes and E edges. Each node has a unique name and knows the names of its direct neighbors. Requested is an asynchronous distributed algorithm which determines a spanning tree (ST) of the graph. Each node performs the same local algorithm, consisting of exchanging messages with the adjacent nodes, and processing received messages. Messages can be transmitted independently in both directions over an edge. After each node completes its local algorithm, it knows which adjacent nodes are in the tree and also which edge leads to a particular node designated as the core of the tree. For the present purposes each node is initially asleep. One or more nodes may wake up spontaneously or upon receiving a wakeup message from a neighbor and will then proceed with their private algorithms. The problem occurs in connection with broadcasting in computer networks. There are many potential control problems for networks whose communication complexity is reduced by having a known spanning tree [11].
A distributed system can adapt to failures in different ways. One way is to temporary halt normal operation and to take some time out to reorganize the system. Such a reorganization is managed by a single node we may call the *leader*. Hence, as a first step in a reorganization the still operative nodes need to *elect* such a leader [12]. Part of the distributed ST algorithm presented below is a *distributed election* of a leader which forms the core of a spanning tree (not necessarily the minimal one). In [11] a solution is presented for distributed *minimum* spanning tree construction using a total number of message passes of at most $5N \log N + 2E$, and a message contains at most one edge weight plus $\log 8N$ bits (logarithms in base 2). Since the core of the spanning tree can be the leader, and below we find the same order of magnitude lower bound for electing a leader, an order $N \log N + E$ lower bound holds for distributed spanning tree (with a designated core) construction too. In fact, the algorithm below finds a *directed spanning tree* which is minimal with respect to the node delays.

Distributed Elections. The problem of decentralized elections has received considerable attention. For general networks [12,10], for complete networks [14,22], and for ring networks e.g [17,4,8,5,19,14,9,27]. In the asynchronous case the basic results are that $O(N \log N)$ message passes suffice for distributed elections in both an undirected and directed ring network [5,19], and $\Omega(N \log N)$ message passes are needed on the average for

both undirected and directed rings [18]. For the synchronous case it was shown that $O(N)$ message passes are sufficient [27,9]. The latter paper shows that the $O(N \log N)$ lower bound holds also for the synchronous case if the distinct names of the processors are used only in comparisons. In a complete graph, distributed elections can be conducted in $O(N \log N)$ message passes, and $\Omega(N \log N)$ is necessary, if the nodes "know" the graph is complete [14]. If the nodes do not "know" the topology of a graph then the election problem is $\Theta(E + N \log N)$ for asynchronous graphs [10,22]..
We exploit a natural property of computer networks, that the ratio of two elapsed time spans anywhere in the network is always finite, in order to obtain a solution which is more frugal in number of message passes than is possible under the harsh assumption of unlimited asynchronicity. For synchronous systems the technique gives better performance than any earlier known method. For synchronous ring networks related ideas have been used in [9,27], and related work for limited asynchronous ring networks in [27]. The limitation on unlimited asynchronism we require is but a minor one which is generally satisfied and which we term "Archimedean asynchronicity".

2. Distributed Systems and Physical Time

In asynchronous distributed systems each processor has its own clock. Although these clocks may not be synchronized, and the clocks may not indicate the same time, there should be some proportion between the clock rates. That is, if an interval of time has passed on the clock for processor A, a proportional period of time has passed on the clock for processor B. This assumption allows us to challenge the $\Omega(N \log N)$ lower bound on the required number of message passes in [3]. In [20] a similar concept "tame" is used advantageously in the context of probabilistic synchronization algorithms.

In the asynchronous networks we consider, the magnitudes of elapsed time satisfy the axiom of Archimedes. The axiom of Archimedes holds for a set of magnitudes if, for any pair a, b of such magnitudes, there is a multiple na which exceeds b for some natural number n. It is called Archimedes' axiom* possibly due to application on a grand scale in *The Sand-Reckoner.*

We assume that the magnitudes of elapsed time, for instance as measured by local clocks amongst different processors or by the clock of the same processor at different times, as well as the magnitudes consisting of communication delays between the sending and receiving of messages, measured in for instance absolute physical time, all together considered as a set of magnitudes of the same kind, satisfy the Archimedean axiom. In physical reality it is always possible to replace a magnitude of elapsed time, of any clock or communication delay, by a corresponding magnitude of elapsed absolute physical time, thus obtaining magnitudes of the same kind. We assume a global absolute time to calibrate the individual clocks; using relative time by having the clocks send messages to one another yields the same effect - for the purposes at hand. If we do not restrict ourselves, so to speak, to Archimedean

* In *Sphere and Cylinder* and *Quadrature of the Parabola* Archimedes formulates the postulate as follows. "The larger of two lines, areas or solids exceeds the smaller in such a way that the difference, added to itself, can exceed any given individual of the type to which the two mutually compared magnitudes belong". The axiom appears earlier as Definition 4 in Book 5 of Euclid's *Elements.*

distributed systems, then the processors in the system may not have any sense of time or have clocks which keep purely subjective time, so that the unit time span of each processor is unrelated to that of another. That is, the set of time units is non-Archimedean by the length of every time unit not being less than a finite times that of any other in the absolute global time scale; or the communication delays having no finite ratio among themselves or with respect to subjective processor clocks. As a consequence distributed elections or any other type of synchronization in a deterministic fashion becomes impossible:

-Any process, pausing indefinitely long with respect to the time-scale of the others, between events like the receiving and passing of a message, and also any unbounded communication delay, effectively aborts activities such as an election in progress. A process can never be sure that it is the only one which considers itself elected.

-Without physical time and clocks there is no way to distinguish a failed process from one just pausing between events.

-A user or a process can tell that a system has crashed only because he has been waiting too long for a response. The nature of time and clocks in distributed systems is discussed in detail in [17,15,16], where the notion of a distributed system, in which elections as described are at all possible, agrees with that of an Archimedean distributed system as defined below. Distributed systems in the sense of *physically* distributed computer networks communicate by sending signed messages and setting timers. If an acknowledgement of safe receipt by the proper addressee is not received by the sender before the timer goes off, the sender sends out a new copy of the message and sets a corresponding timer. This process is repeated until either a proper acknowledgement is received or the sender concludes that the message cannot be communicated due to failures. Thus, clocks and timeouts are necessary attributes of real distributed systems [26] and non-Archimedean time in the system is intolerable outright. Whereas unlimited asynchronism would prevent a system from functioning properly, pure synchronism in a system cannot exist: the clocks of distinct processors drift apart in both indicated time and running speed and have to be resynchronized by algorithms running in Archimedean time as defined below.

Definition. A distributed system is *Archimedean* from time t_1 to time t_2 if the ratio of the time intervals between the ticks of the clocks of any pair of processors, and the ratio between the communication delay between any adjacent pair of processors and the time interval between the ticks of the clock of any processor, is bounded by a fixed integer during the time interval from t_1 to t_2.

Below we treat algorithms for clock synchronization and distributed spanning tree and distributed elections. While the former are time-driven by cause of their very subject matter, the latter are time-driven by design.

3. Clock Synchronization

Assume a Newtonian timeframe in which there exists *one* "real" time. Since we consider a multiprocessor system in isolation, this real time should be local to the system, and system-dependant in a sense explained below. A *clock* is a function C from real time t to logical or indicated time T, or the inverse function $c = C^{-1}$. Viz., there are clocks used for timing events, like keeping user accounts in a CPU, and such clocks are most profitably viewed as mapping real time t to the observed or "clocked" time $T = C(t)$. On the other hand, there are clocks used for instigating events such as a clock used for timeouts or to dial other computers to exchange electronic mail. Those clocks are better viewed as mapping the clock time T to the real time $t=c(T)$. Thus, $|c(T)-c(T')|$ should correspond to the predestined real time interval for a timeout. Capital clock functions are the inverses of the corresponding lower case clock functions.

To *reset* a clock means replacing a a clock function c by another clock function c', and therefore also C by C'. Assume for simplicity that a reset is instantaneous and precise. A reset replaces clock time $C(t)$ by $C'(t)$, with $C'(t)-C(t)=\Sigma$. If $\Sigma>0$ then clock time has disappeared, and actions scheduled for time $T \in (C(t),C'(t)]$ are not initiated unless special measures are taken. If $\Sigma<0$, similarly, actions scheduled for time $T \in (C'(t),C(t)]$ are initiated once more. Worse, timestamps from this location will have a timewarp. For each $t_1 \in (t,c'(T)]$ with $T = C(t)$ (that is, $T_1 \in (C'(t),C(t)]$) the timestamp T_1 will precede any earlier timestamp T_2 which was issued at $t_2 \in (c(T_1),c(T)]$. Thus, the logical order of the timestamps does not correspond with actual order in which they are issued. In a distributed airline reservation system the priority of reservations from even a single branch office may not correspond with reality. Special measures should be taken to prevent such anomalies. One solution [15] to prevent this is resetting clocks only forward ($\Sigma>0$).

We view clocks as continuous increasing functions. We also assume that a clock C is a true physical clock and the *clock rate* drift is not too large:

$$\left|\frac{d^2C(t)}{d^2t}\right| < \kappa \ , \qquad \text{(PC1)}$$

with $\kappa<<1$. If the clock rate drift is larger then we consider the clock faulty.
We consider two clocks C_i and C_j *synchronized* if

$$\left|\frac{dC_i(t)}{dt} - \frac{dC_j(t)}{dt}\right| < \delta \ , \qquad \text{(PC2)}$$

$$|C_i(t)-C_j(t)| < \epsilon \ . \qquad \text{(PC3)}$$

Since we have no way to establish the *physical* true time in an isolated n-processor system, we settle for a *system* true time. By definition, the system time t is given by

$$t = \frac{1}{n}\sum_{i=1}^{n} C_i(t) \ . \qquad \text{(PC4)}$$

If (PC2), (PC3) hold then, for $i=1, \ldots, n$, by (PC4),

$$\left|\frac{dC_i(t)}{dt} - 1\right| < \delta \ , \qquad \text{(PC2')}$$

$$|C_i(t)-t| < \epsilon \ . \qquad \text{(PC3')}$$

Clocks may go run faster and faster or slower and slower without violating property (PC1). If the C's are continuous and differentiable then similar properties with about the same constants will hold for the inverse functions c_1 and c_2.

It is more customary to assume that the clocks may drift, but that the rate at which time is measured by a given clock stays within bounds:

not 10^{-6}?!

$$\left| \frac{\mathrm{d}C(t)}{\mathrm{d}t} - 1 \right| < \theta \ , \qquad \text{(PC1')}$$

with $\theta \ll 1$. (For good crystal clocks one would like $\theta < 10^6$.) Synchronization then requires only PC3, cf [15].. Clock property (PC1') is usually postulated at the outset, and not the result of resynchronisation of clock rates as below. Customarily, clock property (PC1) holds for all $\kappa > 0$, and t indicates the physical true time. We can argue with a physical true time in the system only if its *rate* is not subject to change as is the common assumption. Our problem is more general in making less assumptions about the rate at which time flows. It may be novel to identify the real time with a system true time as does (PC4), but we need to do so.

Finally, the *transmission delay* $D_{ij}(t)$ between processors P_i and P_j is the real time it takes for a message, which is sen~~d~~t off at real time t, to arrive at its destination. We assume that the transmission delay $D_{ij}(t)$ between two processors P_i, P_j is a quantity which is direction independent and does satisfy (PC5).

$$\left| \frac{\mathrm{d}D_{ij}(t)}{\mathrm{d}t} \right| < \nu \ , \qquad \text{(PC5)}$$

with $\nu \ll 1$. If the communication delay drift is larger then we consider the communication link faulty. This latter assumption, of course, may not be satisfied in a real network where connections may deteriorate abruptly. However, with the "nearly infinite" bandwidth communication links which promise to be feasible in the near future, the assumption may well be justified. Moreover, if we cannot say anything about the change of communication delay between ~~all~~ pairs of processors in a network then obviously synchronization is impossible outright.

Below, we first consider the problem for a pair of processors and then generalize the solution.

3.1. Synchronizing Two Processors

We have two processors, P_1 and P_2, with clocks C_1 and C_2. For simplicity we assume that reading the clocks at their location is instantaneous, and similarly the processing of a message and the resetting of a clock. This is justified because by the smoothness of the clock time (PC1), the processor concerned can add/subtract the proper amounts from the quantities concerned. We also do not consider the inevitable small errors which may creep in. For convenience we denote the communication delay D_{12} by just D.

The Z-Algorithm. Each processor P_i $(1 \leqslant i \leqslant 2)$ initiates after each Δ-length elapsed time interval a synchronization round between itself and its direct neighbors (here the other processor). Since the processors can not measure real time, the elapsed time Δ is measured on the local clock C_i. Since according to (PC2') a local clock speed can not be off by more than δ, the real time interval ι in which an synchronization takes place is bounded as follows.

$$\int_x^{x+\iota} (1-\delta)\mathrm{d}t < \int_x^{x+\iota} \mathrm{d}C_i(t) < \int_x^{x+\iota} (1+\delta)\mathrm{d}t \ , \qquad \text{(PA1)}$$

which yields:

$$\frac{|\Delta-\iota|}{\iota} < \delta \ .$$

Another problem is presented by the possibillity in the algorithm below that the clock of a processor is reset during such an interval. We measure the time Δ by running a timer according to the time rate of the current clock, independent of the clock value, thus maintaining (PA1). Synchronizations serve to maintain the synchronized condition (PC2), (PC3) and therefore (PC2') and (PC3'). The value of Δ must guarant~~y~~ee that this happens. Obviously, this can only be achieved if the value of D is not too great relative to the value Δ. In the following algorithm, in each interval ι, each processor resets its clock value once in a resynchronization initiated by itself and once in each resynchronization initiated by each neighbor (the only neighbor in the two-processor case). Moreover, the clock rate is adjusted once to the rate of each neighbor for each resynchronization initiated by that neighbor.

Phase 1. Let a *synchronization* be started by P_1 at time $t_0=c_1(T_0)$. A message stamped with P_1's clock time T_0 is send to processor P_2, and arrives at time $t_1=t_0+D(t_0)$. Clock time on arrival at P_2 is $T_1=C_2(t_1)$. Processor P_2 now knows that P_1 has send off the message at his local time T_0, and that the current clock time at processor P_1 is given by

$$C_1(c_2(T_1)) \geqslant C_1(c_1(T_0)+D(t_0))) \ ,$$

with $D(t_0)>0$.

Phase 2. The message is stamped again, now with P_2's current clock time T_1, and send back to P_1. It arrives there at time $t_2=c_2(T_1)+D(c_2(T_1))$. Clock arrival time at P_1 is $T_2=C_1(t_2)$. Processor P_1 now can deduce various things. In the first place, by (PC5),

$$|D(t_1)-D(t_0)| \leqslant \nu(t_1-t_0) \ .$$

However, P_1 also knows that

$$t_1-t_0 = D(t_0) \ ,$$

and that therefore,

$$(2-\nu)D(t_0) \leqslant D(t_0)+D(t_1) \leqslant (2+\nu)D(t_0) \ .$$

Also,

$$c_1(T_2)-c_1(T_0) = D(t_0)+D(t_1) \ ,$$

and therefore,

$$t_0+\frac{t_2-t_0}{2+\nu} \leqslant c_2(T_1) = t_1 = t_0+D(t_0) \leqslant t_0+\frac{t_2-t_0}{2-\nu} \ .$$

It follows from (PC2') that, at t_0 when (PC2) held,

$$\left| \left[\frac{\mathrm{d}C_1(t)}{\mathrm{d}t}\right]_{t=t_0} -1 \right| < \delta \ ,$$

and therefore, the uncertainty Υ in the clock value of processor P_1 at time t_1 is

$$\begin{aligned}
\pm\Upsilon &= \int_{t_0}^{t_1} \left[\frac{\mathrm{d}C(t)}{\mathrm{d}t} - 1\right]_{t=t_0} + \int_{t_0}^{t} \frac{\mathrm{d}^2 C_1(\tau)}{\mathrm{d}^2\tau}\mathrm{d}\tau\mathrm{d}t \\
&= \int_{\theta=0}^{t_1-t_0} \kappa\theta+\delta\,\mathrm{d}\theta \quad [\theta=t-t_0] \\
&= \left[\frac{\kappa\theta^2}{2}+\delta\theta+\text{constant}\right]_{\theta=0}^{t_1-t_0} . \\
&= \frac{\kappa(t_1-t_0)^2}{2} + \delta(t_1-t_0) .
\end{aligned}$$

We can now express the time $C_1(t_1)$, indicated on the clock of P_1 at real time t_1, in terms of the clock time T_0 at processor P_1 and elapsed real time:

$$C_1(t_1) = T_0 + (t_1-t_0) \pm\Upsilon .$$

Using the estimate for t_1 above,

$$\begin{aligned}
&T_0 + \frac{t_2-t_0}{2+\nu}\left[1-\delta-\frac{\kappa(t_2-t_0)}{2(2+\nu)}\right] \\
&\qquad \leqslant C_1(t_1) \leqslant \\
&T_0 + \frac{t_2-t_0}{2-\nu}\left[1+\delta+\frac{\kappa(t_2-t_0)}{2(2-\nu)}\right]
\end{aligned}$$

By (PC3'), the clock values on clock C_1 at real times t_0, t_2 satisfy $|(T_2-T_0)-(t_2-t_0)| < 2\epsilon$. Therefore, we can express the time with value T_1 on the clock of P_2 (actually $C_1(t_1)$ on the clock of P_1) in terms of the clock values at P_1 as follows:

$$\begin{aligned}
T_1^{\min} &= T_0 + \frac{T_2-T_0-2\epsilon}{2+\nu}\left[1-\delta-\frac{\kappa(T_2-T_0-2\epsilon)}{2(2+\nu)}\right] \\
&\qquad \leqslant C_1(t_1) = C_1(c_2(T_1))) \leqslant \\
T_1^{\max} &= T_0 + \frac{T_2-T_0+2\epsilon}{2-\nu}\left[1+\delta+\frac{\kappa(T_2-T_0+2\epsilon)}{2(2-\nu)}\right]
\end{aligned}$$

We now have an estimate for the difference in clock times between P_1 and P_2 at system time t_1.
(PC3') is satisfied if

$$|T_1^{\min} - T_1| < 2\epsilon , \qquad \text{(PA2)}$$

$$|T_1^{\max} - T_1| < 2\epsilon \ ,$$
$$|T_1^{\min} - T_1^{\max}| < 2\epsilon \ .$$

The algorithm conditions (PA1) and (PA2) induce a set of constraints among Δ, D, δ, ϵ, κ, ν.

We now *reset* the clock of P_1 by replacing C_1 by C'_1:

Case $(T_1^{\min} + T_1^{\max})/2 = T_1$ **then** $C'_1 \leftarrow C_1$;

Case $(T_1^{\min} + T_1^{\max})/2 > T_1$ **then** $C'_1 \leftarrow C_1$; #processor P_2 has to be reset too#

Case $(T_1^{\min} + T_1^{\max})/2 < T_1$ **then**

$$C'_1(t) \leftarrow C_1(t) + T_1 - \frac{T_1^{\max} + T_1^{\min}}{2} \ .$$

By the system time definition (PC4),

$$|C'_1(t_1) - t_1| \leqslant |T_1^{\max} - T_1^{\min}| \ .$$

Phase 3. The message is stamped once more, with P_1's current clock time T_2, and send to P_2. It arrives there at time $t_3 = c_1(T_2) + D(c_1(T_2))$. Clock arrival time is $T_3 = C_2(t_3)$. Processor P_2 can deduce several things. For $i = 1,2,3$, by (PC5):

$$|D(t_i) - D(t_{i-1})| < \nu(t_i - t_{i-1}) \ ,$$
$$t_i = t_{i-1} + D(t_{i-1}) \ .$$

Therefore,

$$|(t_3 - t_1) - (t_2 - t_0)| < \nu(t_2 - t_0) \ .$$

The clock rates may differ by at most δ by (PC2), and therefore:

$$\left| \int_{t_0}^{t_2} dC_1(t) - \int_{t_1}^{t_3 \pm \nu(t_2 - t_1)} dC_2(t) \right| < \delta(t_2 - t_0) \ .$$

Then

$$\begin{aligned} |T_2 - T_0 - T_3 + T_1 \pm \nu(T_2 - T_0)| &< \delta(t_2 - t_0) \\ &\leqslant \delta(T_2 - T_0 \pm 2\epsilon) \ . \end{aligned}$$

This condition is maintained by adjusting the clock rates. Therefore, processor P_2 must be capable of adjusting the rate of P_2's clock to the rate of P_1's clock. We assume that clocks can be reset in the sense that their rate is set to a proportion of the current rate. We *reset* both the clock *rate* and the indicated clock *value* of P_2 by replacing C_2 by C'_2. This clock time is determined by the clock time $T'_2 = C'_1(t_2)$ of the most recent clock C'_1 at P_1, and by $T_2^{\min}$, $T_2^{\max}$. The latter two are determined by the three point bearing using t_1, t_2 and t_3 (just as in Phase 2 the real times t_0, t_1 and t_2 determined $T_1^{\min}$ and $T_1^{\max}$).

If (PC3') is satisfied then

$$|T_2^{\min} - T'_2| < 2\epsilon \ , \qquad \text{(PA3)}$$

$$|T_2^{\max} - T'_2| < 2\epsilon \ ,$$
$$|T_2^{\min} - T_2^{\max}| < 2\epsilon \ .$$

The algorithm conditions (PA1) and (PA3) induce yet another set of constraints among Δ, D, δ, ϵ, κ, ν.

We now *reset* the clock of P_2 by replacing C_2 by C'_2:

Case $(T_2^{\min} + T_2^{\max})/2 = T'_2$ **then**

$$C'_2(t) \leftarrow C_2(t)\frac{T'_2 - T'_0}{T_3 - T_1} \ ;$$

Case $(T_2^{\min} + T_2^{\max})/2 > T'_2$ **then**

$$C'_2(t) \leftarrow C_2(t)\frac{T'_2 - T'_0}{T_3 - T_1}$$

(clock value processor P_1 needs to be reset too);

Case $(T_2^{\min} + T_2^{\max})/2 < T'_2$ **then**

$$C'_2(t) \leftarrow C_2(t)\frac{T'_2 - T'_0}{T_3 - T_1} + T'_2 - \frac{T_2^{\max} + T_2^{\min}}{2} \ ;$$

We can express T'_2 in terms of the clock values at P_2 as follows:

$$T_2^{\min} = T_1 + \frac{T_3 - T_1 - 2\epsilon}{2+\nu}\left[1 - \delta - \frac{\kappa(T_3 - T_1 - 2\epsilon)}{2(2+\nu)}\right]$$

$$\leqslant C_2(t_2) = C_2(c'_1(T'_2))) \leqslant$$

$$T_2^{\max} = T_1 + \frac{T_3 - T_1 + 2\epsilon}{2-\nu}\left[1 + \delta + \frac{\kappa(T_3 - T_1 + 2\epsilon)}{2(2-\nu)}\right]$$

By the system time definition (PC4),

$$|C'_2(t_2) - t_2| \leqslant |T_2^{\max} - T_2^{\min}| \ .$$

Analysis of the Algorithm. In a synchronization interval of length ι, measured as Δ on a clock, we have by (PA1) that $|\Delta - \iota| < \delta\iota$. Let the clocks at the start of the interval (say, $t = x$) be synchronized to within ζ of each others clock values, and their clock rates to within μ. In a Δ-length clock-measured interval, starting at $t = x$ the indicated clock values diverge at most ζ plus twice the divergence from the system time:

$$\text{DVALUE}(\iota) = \zeta + 2\int_{t=x}^{x+\Delta/(1-\delta)} \mu + \int_{\tau=x}^{t} \kappa \, d\tau dt$$

$$= \zeta + 2\left[\frac{\kappa\theta^2}{2} + \mu\theta + \text{constant}\right]_{\theta=0}^{\Delta/(1-\delta)} .$$

$$= \zeta + \frac{2\mu\Delta}{(1-\delta)} + \frac{\kappa\Delta^2}{(1-\delta)^2} \ .$$

In a Δ-length clock-measured interval the clock rates diverge at most:

$$\mathrm{DRATE}(\iota) = \mu + 2 \int_{\tau=x}^{x+\Delta/(1-\delta)} \kappa d\tau \ .$$

$$= \mu + \frac{2\kappa\Delta}{(1-\delta)} \ .$$

In each ι-length real-time interval the clock *values* of P_1 and P_2 are adjusted at least once. Therefore, the new clocks would have been within

$$\zeta \overset{\mathrm{def}}{=} |\, T_i^{\max} - T_i^{\min} \,|$$

of each other at the earlier instant t_i ($i = 1$ or $i = 2$) in this interval. Also, in each such interval the clock *rates* are adjusted to within

$$\mu \overset{\mathrm{def}}{=} |\, T_2^{\max} - T_2^{\min} \,| \,/\, (T_3 - T_1) \ .$$

As long as the clocks are synchronized (PA1) is maintained. To maintain synchronous clocks it is necessary and sufficient that (PA4) (subsuming (PA2) and (PA3)) stays satisfied.

$$\mathrm{DVALUE}(\iota) < \epsilon \ \ \& \ \ \mathrm{DRATE}(\iota) < \delta \ . \qquad \text{(PA4)}$$

It is clear therefore that, for fixed parameters ϵ, δ, κ, ν, if $D(t)$ starts to change then ι, or rather the clocked synchronization interval Δ, must change to maintain (PA4). We can compute from (PC5) how conservative we need to take the fixed parameters above to be able to adjust in time.

3.2. Synchronizing n Processors

Let the network be a graph (V,E) consisting of a set V of n processors connected by a set E of e bidirectional links. This is a simple extension of the two processor case. If we synchronize by having each processor P_i ($i \in \{1,..,n\}$) start a synchronization every Δ clocked time, then each synchronization takes $6e$ messages per $\Delta \pm \epsilon$ real time. Similarly to above, we can determine the constraints on the values of the parameters for which the scheme keeps the n processors synchronized. A more message thrifty method is to have pairwise synchronization between the nodes connected by an edge (i.e., communication link) in a given spanning tree of the network. The number of messages exchanged by the synchronizing Z-algorithm in each period of length $\Delta \pm \epsilon$ then is $6n$.

For such n-processor networks we require (PA4) with $\mathrm{DVALUE}(D\iota) < \epsilon$, $\mathrm{DRATE}(D\iota) < \delta$, with D the *diameter* of the network (of the spanning tree in the spanning tree synchronization, respectively). In the n-processor case ($n > 2$), not only the clock values should be reset only forward, but also the clock rates should only be reset higher. This tactics ensures that, within $D(\Delta \pm \epsilon)$ real time, the highest clock values and the highest clock rates have swept over the entire network.

4. Distributed Spanning Tree and Distributed Elections

We study robust algorithms for distributed elections and spanning tree constructions for arbitrary networks. We analyze in particular the performance for ring networks, networks of bounded degree (e.g., trees, Cube-Connected-Cycles), n-cube networks and complete networks. These algorithms are time-driven and outperform any known algorithm in terms of message passes and passed bits. For both synchronous and asynchronous networks of many topologies, the performance is optimal in these communication complexity measures and can not be improved (in order of magnitude).

4.1. *The Algorithm*

Let G be an undirected connected graph with N nodes and E edges. Assume first that each node has a unique name, say an integer between l and L, $0<l<L$. If the nodes do not have distinct names, but the edges have distinct weights, then let the name of the node be the lexicographical order of the integer weights of the incident edges. There are N nodes, but this is not known to the nodes themselves. Each node contains a clock. The maximal absolute communication delay for one node to communicate with an adjacent neighbor, increased with the largest absolute time for a clocked time unit in the system, is denoted by u. The minimal absolute time for a clocked time unit in the system is denoted by m. Note that u and m may be time dependant. We assume that, from time t_1 to time t_2 we have $0<u/m<s$ for some constant s, that is, the system is Archimedean [27]. The idea for the algorithm is to have a distributed election where the winner is the core of the ST. To be thrifty in message passes, we have to prevent messages originating from future losers to make many message passes before they are eliminated. Election bids have to be stamped by the name of the originator; apart from using the names only for comparisons we also use them to slow down messages from future losers. A program for this Algorithm is given in the Appendix. Below we informally describe the Algorithm, prove it correct, and supply an analysis of the complexity in terms of message passes and passed bits. Subsequently, we study its performance for networks with particular topologies.

Phase 1. Initially each node is asleep. When a node awakes it starts by transmitting a wakeup message to all its neighbors, except the one it received the wakeup message from. A node awakes spontaneously or because it receives a wakeup message. So after the first node awakes it takes $O(uD)$ time to wake up all of the graph, where D is the *diameter* of the graph, that is, the maximum of the set of minimal length paths between pairs of nodes in G. After a node i has been woken up, and sent the wakeup messages over the appropriate edges, it tarries $f(i)$ of its local time units. It then transmits an *election* message, signed with its name, across all edges it is connected to. After having woken up, a node cannot start a new election until after it has received a *victory* message from the elected leader, or after it is itself elected leader. Each node i has a name register containing the least name $i_{\min}$ it has ever seen, originally $i_{\min}=i$, and the edge *link* this name first came from. Originally *link* $=i$. Let the register $i_{\min}$ of node i contain j and let node i be hit by a message M_k arriving over an edge e. If $k \geqslant i$ then $i_{\min}$ does not change and a *reflected* message R_k is sent back over e, cf. Phase 2. If $k<j$ then $i_{\min} \leftarrow k$; *link* $\leftarrow e$. Subsequently, node i counts to $f(k)$ and only then transmits M_k over all edges but the edges over which copies of M_k have arrived at this node. That is, if k has not been replaced in $i_{\min}$ yet by a still lower

name. The basic trick is that for sufficiently fast increasing functions f the other messages are slower than M_l and all other messages together will not account for significantly more message passes than does M_l. Concentrating for the moment on the winning message M_l which traverses *all* edges, we need to argue that the above strategy establishes a unique leader in the network.
Within $uD(1+f(l))$ absolute time all nodes in the graph have received message M_l. Consequently, there is no copy of a message M_i, $i<l$, left in the system. It remains to assess the maximal number of messages which can be passed throughout the system in the meantime. For the passing of a message M_i, originating from a node i, from one node to another we need at least $m\,f(i)$ time units. Each message M_i basically propagates along a private spanning tree with the core in node i. The number of message passes copies of M_i can do depends on the shape of this tree. We assume, that the worst-case forking is bounded by the node degree $d<N$. Therefore, in t time there can be at most

[illegible]

$$\sum_{j=1}^{\lfloor t/mf(j)\rfloor} d(d-1)^{j-1} \ ,$$

message passes of M_i. (Here we take $(d-1)^{-1}=0$.) So the total number of message passes $M_1(G)$ in the system is bounded by

$$M_1(G) \leqslant 2E \ + \sum_{i\in I-\{l\}} \ \sum_{j=1}^{\lfloor \frac{uD(1+f(l))}{mf(i)} \rfloor} d(d-1)^{j-1} \ .$$

where I denotes the set of processor names. For f ~~is~~ a sufficiently fast increasing function, the sum converges, cf. below.
We have now elected a unique leader, but neither the leader nor the others are aware of this yet. Hence, we refine the algorithm as follows.

Phase 2. Each election message M_i leaves a pointer *link* to where it comes from in each pass. Since more than one copy of M_i can hit a node j it has to be resolved which edge is the *link*. This can be done when all copies are present; the node then can also determine all incident edges over which new copies of M_i have to be transmitted.

Lemma 1. *If a node j is hit by more copies of M_i then the last copy hits node j while the first one is still being held.*

Proof. After the retention time of the earlier copy of M_i copies of the message are sen~~d~~t to all nodes from which such a copy was not received yet. By the definition of the algorithm none of the edges in G are used twice to transmit copies of the same message. □

A message M_i retraces its steps to the origin, through the chain of *links*, in three cases. (It retraces its steps in the form of a standard *reflected* message R. For clarity we designate the reflected message of M_i by R_i. It will appear later that this reflected message need carry no identification.)
Case 1. If message M_i hits a node which already contains a message M_i then it is send back in reflected mode R_i.
Case 2. If message M_i hits a node which has but one edge then it is send back in reflected mode R_i.

Case 3. If a node j with $j_{\min}=i$ has send out messages M_i over the set of edges $E(j,i)$, and it has received back copies of the reflected message R_i over all edges from $E(j,i)$, then it sends a message R_i over its edge *link* forthwith.

Lemma 2. *Each node j which sends out the least message M_l over the collection of incident edges $E(j,l)$ receives eventually the reflected message R_l back over all edges in $E(j,l)$.*

Proof. The copies of M_l are stopped only by a node with only one edge or a node connected only to nodes which have already been visited by a copy of M_l, all of which send back R_l over their *links*. The lemma follows by induction. □

Lemma 3. *Let l be the least number identifying a node in the network. In an election according to the above algorithm, only the node numbered l receives its own reflected message R_l back over all its incident edges.*

Proof. First we have to argue that l receives M_l on all incident edges. This follows from Lemma 2. Consequently, the algorithm defines a spanning tree with l as its core, and the reflected messages come from the tips of this tree. Therefore l receives copies of R_l on all incident edges over which it has initially sent M_l.
Second, a node i $(i \neq l)$ does not receive reflected messages on all of its incident edges. Suppose it did, then none of the created copies of M_i ever meets a node through which a lower numbered message has passed. So no created copy of M_i ever is destroyed, except by the reflecting phase, which means that M_i traverses all of the network. But then M_i also hits node l which destroys it: contradiction. □

The reflecting stage can pass the messages along without undue delay. It can also create the converse *link* for every *link* it passes. Thus, each edge in the constructed spanning tree consists of an up*link* and a down*link*. The number of message passes is obviously $M_2(G) = N-1$.

Lemma 4. *A reflected message R_i does not need to contain a number but can consist of a few bits.*

Proof. R_i is eliminated by any election message M_j on its path. This is justified because $j<i$ $(j=i$ is excluded by Lemma 1); otherwise M_j would have been eliminated by the node it just passed through. R_i cannot meet another reflected message R_j in a node since that node would have passed both M_i and M_j at an earlier time, which would result in the previous case. □

When the *least* node l has received messages R_l back from all incident edges, it knows that it is the leader, that there are no other election messages (reflected or not) left in the network, and that all nodes in the network know the identity of the leader. That is, $i_{\min}=l$ for all nodes i in the network.

Phase 3. The leader l now sends a *victory* message along the ST. This stage takes not more than $M_3(N) = N-1$ message passes of a few bits each.

4.1.1. Cost Analysis

Let f be a nondecreasing function with $f(i) \geqslant i$. The overall number of message passes for the algorithm is:

$$\begin{aligned} M(G) &= M_1(G)+M_2(G)+M_3(G) \\ &\leqslant 2(E+N-1) + S \end{aligned}$$

with

$$S = \sum_{i\in I-\{l\}} \sum_{j=1}^{\lfloor \frac{uD(1+f(l))}{mf(i)} \rfloor} d(d-1)^{j-1}$$

more space

$$\leqslant \sum_{i\in I-\{l\}} \frac{d}{d-2}\left((d-1)^{\lfloor \frac{uD(1+f(l))}{mf(i)} \rfloor} - 1\right) \quad [\text{for} \checkmark d\neq 2] \ .$$

We split the range $R=I-\{l\}$ of sum S into R_0 and R_1:

$$R_0 = \{i\in R \mid i > f^{-1}(uDm^{-1}(1+f(l)))\} \ ,$$

$$R_1 = \{i\in R \mid l < i \leqslant f^{-1}(uDm^{-1}(1+f(l)))\} \ ,$$

Then $S = S_0+S_1$ and

$$S_0 = \sum_{i\in R_0} \frac{d}{d-2}\left((d-1)^{\lfloor \frac{uD(1+f(l))}{mf(i)} \rfloor} -1\right) = 0 \ ,$$

$$S_1 \leqslant \frac{d}{d-2}\sum_{i\in R_1}\left((d-1)^{\lfloor \frac{uD(1+f(l))}{mf(i)} \rfloor} -1\right) \ .$$

Here S_1 decreases with faster increasing f. E.g., for $f(i)=2^i$ we obtain

$$S_1 < \frac{d}{d-2} d^{\frac{Du}{m}(\frac{1}{2}+\frac{1}{2^{l+1}}+\epsilon)} \ ,$$

for each $\epsilon>0$. If the algorithm is allowed knowledge concerning the asynchronicity u/m in the network, then we have

$$S_1 < \frac{d}{d-2} d^{D(\frac{1}{2}+\frac{1}{2^{l+1}}+\epsilon)} \ ,$$

for each $\epsilon>0$. If we choose f such that $f(i)m > uD(1+f(l))$ for all $i\in I-\{l\}$ then $S_1=0$ and the message pass complexity $M(G)\leqslant 2(E+N-1)$. This necessitates that f is both very fast increasing and very dependant on the parameters of the network.

The relation between N, E, d and D is given by

$$N \leqslant 1 + d + d(d-1) + \cdots + d(d-1)^{D-1} \approx d^D \ ,$$

and

$$E \leqslant \tfrac{1}{2}dN \approx \tfrac{1}{2}d^{D+1} \ .$$

These inequalities are sharp. Hence, in the borderline cases where inequality can be replaced by equality, we have ~~a~~ topologies for which we can find the least possible upper bounds on S as expressed in terms of N:

$$S < \sum_{i\in I-\{l\}} \frac{d}{d-2}\left(N^{\frac{1}{D}\lfloor \frac{uD(1+f(l))}{mf(i)} \rfloor} -1\right)$$

In general, however, the relation between N, d and D is such that the inequalities can not be replaced by equalities, and the value of S_1 in terms of N is far worse than above. For instance, for ring networks $d^D = 2^{N/2}$ and $E = N$. For Cube-Connected-Cycles, where $E = 1.5N$ (cf. below), $d^D = 3^{2\log N} = N^{3.17}$. For n-dimensional cubes ($n = \log N$, $E = ½N \log N$) $d^D = n^n = N^{\log\log N}$. But for complete networks, $d^D = N - 1$. Yet we shall see in the sequel that, in all these networks, a barely superlinear function f already gives an optimal behavior, which is better than what can be achieved by other known algorithms. Note also, that the performance is achieved by an algorithm which uses only local knowledge.

The *communication* or *bit* complexity of an algorithm measures the total number of bits passed among adjacent nodes in the network. Assume that a message M_i is encoded in $\approx 2\log i$ bits*, by reserving a special pattern to begin and end the message. The wakeup messages, reflected messages and victory messages can then be encoded in 3 bits each. Therefore, the *bit complexity* $B(G)$, the total number of passed bits in the Election, is bounded by:

$$B(G) \leqslant E(3 + 2\log l) + 6N - 6 + \sum_{i \in I - \{l\}} 2\log i \sum_{j=1}^{\left\lfloor \frac{uD(1+f(l))}{mf(i)} \right\rfloor} d(d-1)^{j-1} \ .$$

4.1.2. Minimum Spanning Tree

It is not difficult to see that the constructed spanning tree is, for practical purposes, a minimum spanning tree insofar as the delay in the processing nodes is concerned.

4.1.3. Synchronous Systems

For synchronous systems we assume that the message delivery time is negl~~ed~~igible with respect to the time used by the process clocks; in particular that the "asynchronicity coefficient" u / m is equal to 1. This means that for such systems, if we choose f such that

$$f(i) > D.(1 + f(l))$$

for all $i \in I - \{l\}$ then the message pass complexity $M(G)$ and the bit complexity $B(G)$ are

$$M(G) \leqslant 2(E + N - 1) \quad \& \quad B(G) \leqslant E(2\log l\ +\ 3)\ +\ 6(N-1)\ .$$

Even if f does not rise quite so fast, we shall see below that the contributions from the future losers in the election (the sum in the expressions for $M(G)$ and $B(G)$ in the previous section) does not contribute much. Indeed, for most network topologies investigated below it appears that, for synchronous networks, for $f(i) = 2^i$ or $f(i) = 3^i$ the total contribution from future losers is not essentially more than the contribution from the single winner.

* For convenience, we always simply denote '$\lceil \log_2(i+1) \rceil + 1$' by '$\log i$'.

4.2. Ring Networks

On the macroscopic scale of computer networks a popular configuration is a physical or virtual ring [21]. An example of a multiprocessor system organized in a ring is the University of Maryland's ZMOB, a cabinetsized 256 processor ring network [24]. Ring networks are often one-directional. For one-directional ring networks, since the single leaf of the directed spanning tree is the node adjacent to the root, the reflection phase can be dropped setting $M_2=0$. This only marginally reduces the total. We prefer to stick to the general method above for the estimate. For a ring network we have $D=N/2$, $E=N$ and $d=2$. Therefore,

$$M(G) \leqslant 4N-2+\sum_{i\in I-\{l\}} 2\left\lfloor \frac{uN(1+f(l))}{2mf(i)} \right\rfloor .$$

In case f is superlinear, that is, $f \in \Omega(i^{1+\epsilon})$ with $\epsilon>0$, the sum converges.
As example, let $f(i)=i^{1+\epsilon}$, $\epsilon>0$. Then (to obtain the 5th line split the sum $\sum_{i=l+1}^{\infty}$ in l subsums $\sum_{k[j]=1}^{\infty}$ with $k[j]=i \bmod l$ $(0\leqslant j<l)$, and majorize the subsums by an integral):

$$\begin{aligned}
M(G) &\leqslant 4N-2+\frac{uN(1+f(l))}{m}\sum_{i\in I-\{l\}}\frac{1}{f(i)} \\
&< 4N-2+\frac{uN(1+f(l))}{mf(l)}\sum_{i=l+1}^{\infty}\frac{f(l)}{f(i)} \\
&\leqslant 4N-2+\frac{uN}{m}(1+l^{-(1+\epsilon)})\sum_{i=l+1}^{\infty}\left[\frac{l}{i}\right]^{1+\epsilon} \\
&< 4N-2+\frac{uN}{m}(l+l^{-\epsilon})\int_{i=1}^{\infty} i^{-(1+\epsilon)}\mathrm{d}i \\
&= 4N-2+\frac{uN(l+l^{-\epsilon})}{m\epsilon} . \\
&\leqslant 4N-2+\frac{uN(l+1)}{m\epsilon} \quad (l\geqslant 1).
\end{aligned}$$

The number of passed *bits* is also low. Recall, that a message M_i is encoded in $\approx 2\log i$ bits, by reserving a special pattern to begin and end the message. The wakeup messages, reflected messages and victory messages can then be encoded in 3 bits each. The *bit complexity* $B(G)$, for $f(i)=i^{1+\epsilon}$ $(\epsilon>0)$ then turns out to be (for the second line split the sum in l subsums as above and notice that $1+\log_l i=\log li/\log l$)

$$\begin{aligned}
B(G) &< 3(3N-2)+2N\log l+\frac{Nu(l^{1+\epsilon}+1)}{m}\sum_{i=l+1}^{\infty}(2\log i)i^{-(1+\epsilon)} \\
&\leqslant 9N-6+2N\log l+\frac{2Nu(l+l^{-\epsilon})\log l}{m}\int_{i=1}^{\infty}(1+\log_l i)i^{-(1+\epsilon)}\mathrm{d}i \\
&< 9N-6+2N\log l+\frac{6Nu(l+1)\log l}{m\epsilon} \quad (l\geqslant 1) .
\end{aligned}$$

For different functions f we obtain ever better performance, as tabulated below.

$f(i) =$	worst-case message passes	worst-case passed bits
constant	$\Theta(N^2)$	$O(N^2 \log L)$
i^ϵ ($\epsilon > 1$)	$O(\frac{Nul}{m\epsilon})$	$O(\frac{Nul \log l}{m\epsilon})$
2^i	$O(\frac{Nu}{m})$	$O(\frac{Nu \log l}{m})$
$(2u/m)^i$	$\Theta(N)$	$\Theta(N \log l)$

Table. N is the number of processors in the ring network; u is the maximum unit time among the processors increased with the greatest communication delay between adjacent processors; m is the minimum unit time among the processors; l is the least integer name of a processor; L is the greatest integer name of a processor.

Therefore, if f increases fast enough, the number of messages/bits needed for an election in the ring network hits the rock bottom. In particular, for $f(i) = 2^i$ we actually have $M(G) \leqslant 4N + um^{-1}N$, and for $f(i) = (2um^{-1})^i$ we have $M(G) \leqslant 5N$ and $B(G) \leqslant 9N + 4N \log l$. For the particular case of elections in a ring (the spanning tree construction is then less relevant) we can dispense with the reflected messages [27]. In the latter reference we also express the average number of message passes or passed bits used in an this manner of distributed election in a ring.

4.2.1. *Synchronous Case*

In the *synchronous* case the results are simply those in the Table above with u/m set to 1. In particular, for $f(i) = 2^i$ we have $M(G) \leqslant 6N$ and $B(G) \leqslant 9N + 6N \log l$.

4.3. *Networks of Bounded Degree*

If all nodes in the network have degree bounded by a constant $d > 2$ then the general solution needs:

$$M(G) \leqslant 2(E + N - 1) + \sum_{i \in I - \{l\}} \sum_{j=1}^{\lfloor \frac{uD(1+f(l))}{mf(i)} \rfloor} d(d-1)^{j-1} .$$

This is the case for tree networks and X-tree networks, but also for various *fast permutation* networks like the Cube-Connected-Cycles, the shuffle-exchange network, or the butterfly (FFT) network, cf [2].. An example of a multiprocessor systems along these lines is the Bolt, Beranek & Newman Butterfly; 128 processors in a butterfly network. Another example is New York University's ULTRAcomputer [23, 13], a shuffle-exchange multiprocessor. The most easily explained type is the Cube-Connected-Cycles (CCC) network. It consists of basically an n-dimensional cube, with each n-degree corner node replaced by a n-node cycle. Each node on such a cycle has one edge incident of the n edges which were incident on that corner before. The degree d of all nodes in the CCC network is therefore 3, $E=1.5N$ and the diameter D is $2\log N$. Therefore we obtain:

$$M(G) \leqslant 5N-2 + 3 \sum_{i \in I-\{l\}} (2^{\lfloor \frac{2u(1+f(l))\log N}{mf(i)} \rfloor} - 1) \ .$$

For the values of i for which the exponent in the sum is 0 there is no contribution by the corresponding term. Therefore we may assume that

$$\frac{2u(1+f(l))\log N}{mf(i)} \geqslant 1 \ .$$

Then, we can majorize $M(G)$ by

$$M(G) < 5N-2 + 3 \sum_{i \in R} (2^{\lfloor \frac{2u(1+f(l))\log N}{mf(i)} \rfloor} - 1) \ ,$$

with

$$R = \{ i \mid l+1 \leqslant i \leqslant f^{-1}(2um^{-1}(1+f(l))\log N) \} \ .$$

4.3.1. $f(i)=i^2$
Split the sum over range R in subsums over disjoint subranges R_1, R_2:

$$R_1 = \{ i \mid l+1 \leqslant i \leqslant 2l+1 \}$$
$$R_2 = \{ i \mid 2l+1 < i \leqslant f^{-1}(2um^{-1}(1+f(l))\log N)$$

Then,

$$M(G) < 5N-2 + 3(l+1)[N^{\frac{2u(1+l^2)}{m(l+1)^2}} + (N^{\frac{u}{m}} 2um^{-1} \log N)^{½}] \ .$$

The *bit* complexity $B(G)$, under the same encoding scheme as in the previous section, is estimated by

$$B(G) < 10.5N + 3N \log l - 6 + 6(l+1) \log l \ N^{2um^{-1}(1+l^2)(l+1)^{-2}} +$$
$$6(l+1)\sqrt{2um^{-1}\log N} \ \log((l+1)\sqrt{2um^{-1}\log N})N^{\frac{u}{2m}} \ .$$

4.3.2. $f(i)=3^i$.
The influence of the choice of l disappears since the sum is only over $um^{-1}\log N$ terms. The range R is now

$$R = \{ i \mid l+1 \leqslant i \leqslant \log_3(2um^{-1}(1+3^l)\log N) \} \ .$$

$$
\begin{aligned}
M(G) &\leqslant 5N-2 + 3\sum_{i\in R}\left(2^{\left\lfloor \frac{2u(1+3^{l})\log N}{m3^{i}} \right\rfloor}-1\right) . \\
&< 5N-2 + 3N^{\frac{2u(1+3^{l})}{m3^{l+1}}} \log_3(um^{-1}\log N) \\
&< 5N-2 + 3N^{\alpha um^{-1}} \quad [0<\alpha<1] .
\end{aligned}
$$

Similarly,

$$
\begin{aligned}
B(G) &< 10.5N+3N\log l-6 + 6N^{\alpha um^{-1}}\log l\log_3(2um^{-1}\log N) \\
&< 10.5N+3N\log l + 6N^{\beta um^{-1}}\log l \quad [0<\beta<1] .
\end{aligned}
$$

4.3.3. $f(i) = 3^{um^{-1}}$

Now not only the effect l but also the effect of u/m is removed from the complexity bounds:

$$M(G) < 8N \quad \& \quad B(G) < 10.5N + 9N\log l .$$

4.3.4. Synchronous Case

Let $f(i)=i^2$. In the *synchronous* case ($u=m$) with also $l=1$ we therefore have

$$M(G) < 11N-2+6(2N\log N)^{½} .$$

$$B(G) < 25.5N-6+6(2N\log N)^{½}\log(2\log N)^{½} .$$

Let $f(i)=3^i$. Then, with $u=m$,

$$M(G) < 8N \quad \& \quad B(G) < 10.5N + 9N\log l .$$

4.4. n-Cubes

The network forms an n-cube, that is, there are $N=2^n$ nodes. Each node is incident on n edges. So the number of edges $E=n2^{n-1}=(N\log N)/2$, the degree of each node is $d=\log N$, and the diameter $D=\log N$. An example of a multiprocessor system of that nature is Caltech's Cosmic Cube [25], consisting of 64 processors arranged in a binary 6-cube. Here the effect of many distinct paths leading to the same nodes becomes more pronounced. In particular, the number of nodes which are i edges or less distant from a given node is $\sum_{j=0}^{i}\binom{\log N}{j}$. The number of edges in between is at most $\sum_{j=1}^{i} j\binom{\log N}{j}$. Therefore,

$$M(G) < 2(N\log N + N-1) + \sum_{i\in I-\{l\}} \sum_{j=1}^{\left\lfloor \frac{u\log N(1+f(l))}{mf(i)} \right\rfloor} j\binom{\log N}{j}$$

For the solution the following inequality is advantageous.

$$\sum_{j=0}^{\frac{1}{3}N-1} j \binom{N}{j} < 2 \binom{N}{\frac{1}{3}N} .$$

We split the sum

$$S = \sum_{i \in I-\{l\}} i \sum_{j=1}^{\lfloor \frac{u \log N(1+f(l))}{mf(i)} \rfloor} j \binom{\log N}{j}$$

in three parts S_0, S_1 and S_2. In S_0 we collect the message passes for $i \in R_0$,

$$R_0 = \{ i \mid \lfloor \frac{u \log N(1+f(l))}{mf(i)} \rfloor < 1 \} ,$$

that is, messages which are not passed at all. In S_1 we collect the message passes for $i \in R$ for which the messages may pass at least once and so that we can use the binomial inequality above.

$$R_1 = \{ i \mid 1 < \lfloor \frac{u \log N(1+f(l))}{mf(i)} \rfloor < \frac{\log N}{3} \} .$$

In S_2 we group the message passes of the messages M_i which may pass often. That is, $i \in R_2$ such that

$$R_2 = \{ \lfloor \frac{u \log N(1+f(l))}{mf(i)} \rfloor \geqslant \frac{\log N}{3} \} ,$$

which is the case for a small number of i's for which, moreover, $i - l$ is small. Then:

$$S_0 = 0 .$$

$$S_1 = \sum_{i \in R_1} \sum_{j=1}^{\lfloor \frac{u \log N(1+f(l))}{mf(i)} \rfloor} j \binom{\log N}{j}$$

$$< \sum_{i \in R_1} 2 \binom{\log N}{\frac{1}{3}\log N}$$

$$< [f^{-1}(um^{-1}(1+f(l))\log N) - f^{-1}(3um^{-1}(1+f(l)))] N$$

$$S_2 = \sum_{i \in R_2} \sum_{j=1}^{\lfloor \frac{u \log N(1+f(l))}{mf(i)} \rfloor} j \binom{\log N}{j}$$

$$< \sum_{i \in R_2} N \log N$$

$$< (f^{-1}(3um^{-1}(1+f(l))) - l) N \log N .$$

Therefore,

$$M(G) < 2(N\log N + N-1) + S_1 + S_2 \ .$$

The number of bits is given by

$$B(G) < 2N\log N(\log l + 3) + 6(N-1) + S'_1 + S'_2 \ ,$$

with

$$S'_1 < 2S_1 \log(f^{-1}(um^{-1}(1+f(l))\log N) \ .$$

$$S'_2 < 2S_2\log\left[\frac{S_2}{N\log N} + l\right] ,$$

In the Table below we give the S_1 component and the S_2 component of $M(G)$, for different choices of f.

$f(i)=$	$S_2 <$	$S_1 <$
i	$(3um^{-1}(1+l)-l)N\log N$	$um^{-1}(1+l)N\log N$
i^2	$(\sqrt{3um^{-1}(1+l^2)}-l)N\log N$	$N\sqrt{um^{-1}(1+l^2)\log N}$
2^i	$N\log N\ \log(3um^{-1})$	$N\log\log N$
$(2u/m)^i$	$2N\log N$	$N\log\log N$

Table. N is the number of processors in the n-cube, u is the maximum unit time among the processors increased with the greatest communication delay between adjacent processors; m is the minimum unit time among the processors; l is the least integer name of a processor.

4.4.1. Synchronous Case

With $u/m=1$ we have, very roughly, the following upper bounds.

For $f(i)=i^2$:

$$M(G) < (2l+5)N\log N + 2N \ ,$$

$$B(G) < 2(5+\log l + 2l\log l)N\log N + 6N \ .$$

For $f(i)=2^i$:

$$M(G) < 4N\log N + 2N \quad \& \quad B(G) < 2N\log N(4\log l + 3) + 6N \ .$$

4.5. Logarithmic Branching Trees

The network is an N node tree with $N-1$ edges and all internal nodes of degree $d=\log N$, diameter $D=2\log\log N$ and of approximately $(\log N)/(\log\log N)$ levels.

Then

$$M(G) \leqslant 2(E+N-1) + \sum_{i\in I-\{l\}} \sum_{j=1}^{\left\lfloor \frac{2u(1+f(l))\log N}{mf(i)\log\log N} \right\rfloor} \log^j N$$

$$< 2(E+N-1) + \sum_{i\in I-\{l\}} 2(\log N)^{\frac{2u(1+f(l))\log N}{mf(i)\log\log N}}$$

$$< 2(E+N-1) + 2\sum_{i\in I-\{l\}} N^{\frac{2u(1+f(l))}{mf(i)}} \quad .$$

For $f(i)=2^i$ this gives a crude upper bound

$$M(G) < 2(2N-2) + 2N^{\frac{2u}{m}(\frac{1}{2}+\frac{1}{2^{l+1}}+\epsilon)} \qquad [\epsilon>0] \ .$$

In the synchronous case for $f(i)=2^i$, as well as in the asynchronous case with $f(i)=(2u/m)^i$, we can set $u/m=1$ in this formula for S. For such a very fast branching network, the efficiency of the method depends more critically on the choices of processor names and function f than the previous examples. This effect culminates in the fastest branching networks: complete networks.

4.6. Complete Networks

If each node in the network is connected with every other node then we have a complete network. Here we have N nodes of degree $d=N-1$ each, $E=N(N-1)/2$ edges and the diameter $D=1$. Owing to the sharp rise of the number of point-to-point connections with increasing N this is not a popular topology for large networks based on message passing. Since the diameter of the network is a constant, which means that the node degree $d(G)$ must be linear in N, the method is very sensitive to the u/m ratio, and performs poorly in the worst case. More interesting is to analyse the more or less synchronous case.

So let u/m be a constant near 1. If $f(i)>um^{-1}(1+f(l))$ then the wakeup message from any node i wakes up node l so quick, that the election message from l has reached all other nodes before their election messages are transmitted. Pick a function f such that this happens, e.g., $f(i)=2^i$. Then

$$M(G) = 2(E+N-1) = N^2+N-2 \ ,$$

$$B(G) = 0.5(N^2-N)(2\log l+3) + 6N-6 \ .$$

4.7. Optimality

For both distributed elections and distributed spanning tree construction, an obvious lower bound on the message pass complexity $M(G)$ is formed by the number of edges E of the network G. Each untraversed edge in the network could have been be the only edge leading to a particular node thus preventing both valid elections and spanning trees. Similarly, the bit complexity $B(G)$ of the elections is bounded below by $E \log l$, since the name l of the choosen Leader has to be communicated to all nodes in the network.

For most *synchronous* networks we have examined, the upper bounds come within a (small) constant multiplicative factor of these lower bounds for functions $f(i)=c^i$ ($c=2,3$).

For most *asynchronous* networks we have examined, the upper bounds come within a (small) constant multiplicative factor of these lower bounds for functions $f(i)=(cu / m)^i$ ($c=2,3$). Here u / m can be viewed as the asynchronicity factor which typically should be low, e.g. $u / m < 2$.

The algorithm is less suited for networks which branch out very quickly. For the logarithmic branching tree we need faster increasing f, or the worst case behavior in message passes may rise to order N^2, while for the complete network the worst case behavior can rise to order N^3 message passes.

The time complexity $T(G)$ of the solutions in a network G is,

$$T(G) < 3uD + f(l)uD \quad ,$$

where we can take uD to be the maximum of the set of minimal communication delays between any pair of nodes in the network. Therefore, $T(G)$ is within a multiplicative factor $f(l)+3$ of the optimal time solution as well. For the considered functions $f(l)$ this is very good if l is low. Obviously, in a dynamic changing network, where nodes which are inserted have to choose an as yet nonexisting name, node insertion can only decrease l. If a node which is taken out exchanges names with a remaining neighbor, in case that neighbors name is higher, then node deletion also does not increase l. Hence, under this strategy l tends to decrease as N increases, which improves the time performance. Yet, if the processors get real names, like “mcvax” or “ihnp4”, the translation into numbers is unlikely to yield a small number for the least name. Hence, to obtain a reasonable time performance, we should hash the names to positive integers to be used as names in the algorithm. This requires a hash function with properties not easily met elsewhere. Viz.,

1. All names should be mapped to different integers. Actually, for correctness of the algorithm it is only required that the name mapping to the least integer be unique. Mapping more names to the other integers only degrades performance.
2. The least integer mapped to should be small, preferably 1.
3. The performance of the algorithm is best if the range of integers the names are mapped into is as large as possible, and the images of the names are evenly distributed over this range.

Finally, the algorithm is robust in the sense that it always works, in networks of any topology, without knowing the topology of the network, and under any asynchronity in the network. Yet it performs often as well or better than known algorithms in synchronous networks where the topology is known to each node. Amongst algorithms which do not assume any global knowledge of the network, it seems the most efficient one known, in terms of message passes and passed bits. With increasing function $f(i)$ and decreased u/m and l, the performance of the method smoothly improves at accelerated speed.

5. Appendix

Let G be a network consisting of a set N of processor nodes and a set E of bidirectional communication links between pairs from N. Initially all processor nodes are functioning happily in their normal mode which we, for the present purposes, call being *asleep*. Suddenly, one or more *awake*, that is, become aware that an election is due. Between this time and the time the Leader is determined, and all processors have been notified thereof, any processor which awakes executes the Protocol below. Processes awake spontaneously, and in any event when they receive a *wakeup* message from a neighbor. On notification of a successful election by a *victory* message a process falls asleep again.
The local node on which the Protocol runs is node i. Node $i_{\min}$ is the currently and locally designated winner of the election and *link* is the edge the first election message of the current winner arrived over. The set of edges $E(i)$ denotes the set of edges incident on node i. The set of edges $R(i)$ is, initially, the set of edges incident on node i over which no copies of the currently winning election bid have yet arrived. After the appropriate wait, node i sends out copies of the currently winning election bid over the nodes in $R(i)$. Subsequently, $R(i)$ is the set of edges incident on node i on which the corresponding reflected messages R have not yet arrived. The set of edges $ST(i)$ is the set of edges incident on node i in the currently constructed spanning tree. There are four types of messages: *wakeup* messages W, *election* messages M_j (stamped with the name j of the bidder), *reflected* messages R and *victory* messages V. To distinguish these messages we can code them in 3 bits each plus the attached $2\log j$ bits for coding j in each election message M_j $(l \leqslant j \leqslant L)$.

Protocol to be executed when processor i awakes.

```
Send messages $W$ to all adjacent nodes except the one from which a $W$ message waking $i$ came
from; Set $i_{\min}$ and *link* equal to $i$ and set *timer* equal to $f(i)$;
REPEAT IN EACH (LOCAL) TIME "UNIT":
for $i=1$ step 1 until $\#E(i)$
  do
    Read incoming message $M$ (possibly $M=\varnothing$) from the next edge $e$ in $E(i)$;
    if I am awake and timer $=0$ and $M=R$ and $R(i)-\{e\}=\varnothing$ then send a $V$ message over all
    edges in $E(i)$: the election is finished; #for all nodes $j$ in $G$ currently $j_{\min}=i$#
    if I am awake and $M=V$ then
    begin
      Leader $\leftarrow i_{\min}$;
      send $V$ over all edges in $ST(i)-\{link\}$ and go to sleep
    end
    if I am awake and $M=R$ then
    begin
      if timer = 0 then
      begin
        $R(i) \leftarrow R(i) - \{e\}$; $ST(i) \leftarrow ST(i) \cup \{e\}$;
        if $R(i) = \emptyset$ then send $R$ over link;
      end
      if I am awake and $M = M_j$ then
      begin
        if $j < i_{\min}$ then
        begin
          $i_{\min} \leftarrow j$ ; $link \leftarrow e$; $ST(i) \leftarrow \{link\}$;
          $R(i) \leftarrow E(i) - \{e\}$; $timer \leftarrow f(i_{\min})$
          end
        if $j = i_{\min}$ or $E(i) = \{e\}$ then send $R$ over edge $e$
        if $j \geqslant i_{\min}$ then
        begin
          $timer \leftarrow timer - 1$;
          if $timer = 0$ then send $M_{i_{\min}}$ over the edges in $R(i)$;
        end
      end
    end
  od
```

REFERENCES

[1] "Several papers," in Proceedings 3rd ACM Symposium on Principles of Distributed computing.

[2] Broomel, G. and J.R. Heath, "Classification categories and historical development of circuit switching topologies," *ACM Computing Surveys*, vol. 15, pp.95-133, 1983.

[3] Burns, J.E., "A formal model for message passing systems", Technical Report No. 91, Computer Science Department, Indiana University, May 1980.

[4] Chang, E. and R. Roberts, "An improved algorithm for decentralized extrema-finding in circular configurations of processes," *Communications of the Ass. Comp. Mach.*, vol. 22, pp.281 - 283, 1979.

[5] Dolev, D., M. Klawe, and M. Rodeh, "An O(n log n) unidirectional distributed algorithm for extremafinding in a circle," *Journal of Algorithms*, vol. 3, pp.245-260, 1982.

[6] Dolev, D., J. Halpern, B. Simons, and H.R. Strong, "Fault-tolerant clock synchronization," in 3rd ACM Symposium on Principles of Distributed Computing (1984).

[7] Dolev, D., J. Halpern, and H.R. Strong, "On the possibility and impossibility of achieving clock synchronization," pp. 504-511 in 16th ACM Symposium on Theory of Computing (1984).

[8] Franklin, R., "On an improved algorithm for decentralized extrema finding in circular configurations of processors," *Communications of the Ass. Comp. Mach.*, vol. 25, pp.336-337, 1982.

[9] Frederickson, G.N. and N.A. Lynch, "The impact of synchronous communication on the problem of electing a leader in a ring," pp. 493-503 in Proceedings 16th Annual ACM Symposium on Theory of Computing (1984).

[10] Gallager, R.G., "Finding a leader in a network in O(e)+O(n log n) messages", Internal Memo, MIT, 1979.

[11] Gallager, R.G., P.A. Humblet, and P.M. Spira, "A distributed algorithm for minimum weight spanning trees," *ACM Transactions on Programming Languages and Systems*, vol. 5, pp.66-77, 1983.

[12] Garcia-Molina, H., "Elections in a distributed computing system," *IEEE Transactions on Computers*, vol. C-31, pp.48-59, 1982.

[13] Gottlieb, A., R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolf, and M. Snir, "The NYU Ultracomputer - Designing a MIMD shared memory parallel computer," *IEEE Transactions on Computers*, vol. C-32, pp.175-190, 1983.

[14] Korach, E., S. Moran, and S Zacks, "Tight lower and upper bounds for distributed algorithms for a complete network of processors," in 3rd Annual ACM Symposium on Principles of Distributed Computing (1984).

[15] Lamport, L., "Time, clocks, and the ordering of events in a distributed system," *Communications of the Ass. Comp. Mach.*, vol. 21, pp.558-565, 1978.

[16] Lamport, L. and P.M. Melliar-Smith, "Synchronizing clocks in the presence of faults," *J. Ass. Comp. Mach.*, vol. 32, pp.52-78, 1985.

[17] LeLann, G., "Distributed systems - Towards a formal approach," pp. 155-160 in Information Processing 77, ed. B. Gilchrist, North Holland, Amsterdam (1977).

[18] Pachl, J., E. Korach, and D. Rotem, "Lower bounds on distributed maximum-finding algorithms," *J. Ass. Comp. Mach.*, vol. 31, pp.905-919, 1984.

[19] Peterson, G.L., "An O(n log n) unidirectional algorithm for the circular extrema problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, pp.758-762, 1982.

[20] Reif, J.H. and P.G. Spirakis, "Real-time synchronization of interprocess communications," *ACM Transactions on Programming Languages and Systems*, vol. 6, pp.215-238, 1984.

[21] Saltzer, J.H., K.T.Pogran, and D.D. Clark, "Why a ring?," *Computer Networks*, vol. 7, pp.223-231, 1983.

[22] Santoro, N., "Sense of direction, Topological Awareness and Communication Complexity," *SIGACT News*, vol. 16, no. 2, pp.50-56, 1984.

[23] Schwartz, J.T., "Ultracomputers," *ACM Trans. on Programming Languages and Systems*, vol. 2, pp.484-521, 1980.

[24] Seitz, Ch.L., "Concurrent VLSI architectures," *IEEE Transactions on Computers*, vol. C-33, pp.1247-1265 , 1984.

[25] Seitz, Ch.L., "The cosmic cube," *Communications of the Ass. Comp. Mach.*, vol. 28, pp.22-33, 1985.

[26] Tanenbaum, A.S., *Computer Networks*. Englewood Cliffs, New Jersey:Prentice-Hall, 1981.

[27] Vitányi, P.M.B., "Distributed elections in an Archimedean ring of processors," pp. 542-547 in Proceedings 16th ACM Symposium on Theory of Computing (1984).