Network Orientation

Gerard Tel

RUU-CS-91-8
March 1991

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

Network Orientation

Gerard Tel

Technical Report RUU-CS-91-8
March 1991

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

Network Orientation

Gerard Tel*
Department of Computer Science, University of Utrecht,
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
(Email: gerard@cs.ruu.nl)

Abstract

This paper analyses how the symmetry of a processor network influences the existence of a solution for the network orientation problem. The orientation of cliques, hypercubes and tori is the problem of assigning labels to each link of each processor, in such a way that a sense of direction is given to the network. In this paper the problem of network orientation for these two topologies is studied under the assumption that the network contains a single leader, under the assumption that the processors possess unique identities, and under the assumption that the network is anonymous. The distinction between these three models is considered fundamental in distributed computing.

It is shown that orientations can be computed by deterministic algorithms only when either a leader or unique identities are available. Orientations can be computed for anonymous networks by randomized algorithms, but only when the number of processors is known. When the number of processors is not known, even randomized algorithms cannot compute orientations for anonymous processor networks.

Lower bounds on the message complexity of orientation and algorithms achieving these bounds are given.

1 Introduction

In this paper the problem of orienting processor networks is considered for cliques, hypercubes and tori. The orientation problem concerns the assignment of different labels ("directions") to the edges of each processor, in a globally consistent manner. The label of an edge indicates in which direction in the network this edge leads, and this information is useful for purposes of routing and traversal of networks.

The results obtained for this problem serve to illustrate a number of fundamental results in distributed computing obtained during the last decade. The paper treats issues of symmetry in networks of processors in depth; the results in this area have to do with deterministic versus randomized algorithms, election and name assignment, and the computational power of anonymous networks. The paper includes brief discussions of some of the most challenging problems in distributed computing, including fault–tolerance, synchronism, and termination detection.

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

1.1 Computing Orientations

It was demonstrated by Santoro [San84] that the availability of an orientation decreases the message complexity of important computations in networks of several topologies. (A formal definition of orientations is deferred to Subsection 2.1). For example, an $\Omega(N \log N)$ lower bound was proved (see Korach *et al.* [KMZ84]) on the message complexity of electing a leader in an unoriented clique of N processors. For oriented cliques an algorithm using $O(N)$ messages exists; see Loui *et al.* [LMW86]. Kranakis and Krizanc's algorithm for computing boolean functions on hypercubes [KK90a] assumes that an orientation of the hypercube is available. Similarly, Beame and Bodlaender's algorithm for computing boolean functions on torus networks [BB89] assumes that an orientation of the torus is available. For both network topologies it is not known, whether the same complexity (for computing arbitrary boolean functions) is achievable in unoriented networks. It is known though [KK90a] that the collection of computable functions is larger for oriented networks.

Surprisingly, although the importance of orientations is well known, only few papers have addressed the question how orientations can be computed in networks where no orientation is available. Peterson [Pet85] has presented an efficient election algorithm for oriented tori, and claimed that this algorithm can be adapted to work on unoriented tori, thus avoiding the question of computing an orientation. Korfhage and Gafni [KG85] have presented an algorithm to orient *directed* tori. The orientation problem for tori was also studied by Syrotiuk *et al.* [SCP93]; see the end of Subsec. 2.4. (In this paper only undirected tori are considered.) There has been considerable interest in the problem of orienting a ring network [IJ93, SP87, CS92].

1.2 Network Symmetry

A fundamental notion in the study of distributed algorithms is the issue of the required *symmetry* of a solution. In this paper the orientation problem is studied under three different symmetry assumptions: all processors execute a different algorithm, namely a standard algorithm parametrized by the name of a processor (*named* network); one processor executes a different algorithm, all others execute the same algorithm (*leader* network); all processors execute the same algorithm (*anonymous* network). (These assumptions will be presented more precisely in Subsection 2.2.) The class of deterministically computable functions is the same for leader networks and for named networks. It was shown by Angluin [Ang80] that anonymous networks can deterministically compute strictly less functions than leader networks. It was later shown by Itai and Rodeh [IR81] that anonymous networks can simulate leader networks with a randomized algorithm when the number of processors is known. Also, when the number of processors is not known, anonymous networks can randomizedly compute strictly less function than leader networks. The results in this paper illustrate these fundamental results by analyzing the solvability of orientation as a function of the required symmetry.

About this paper. This paper is organized as follows. In Section 2 the orientation problem and the three symmetry models are defined, lower bounds are proved, and impossible cases are identified. In Section 3 the problem is considered for cliques, in Section 4 the problem is considered for hypercubes, and in Section 5 for tori. In Section 6, which is of a very technical nature, a different characterization of orientations is presented. Sec-

tion 7 contains conclusions and general remarks, and discusses some important problems in the area of distributed computing.

2 Preliminary Results

In Subsection 2.1 the formal definitions of the considered network topologies are given, and the formal definitions of labelings and orientations. In Subsection 2.2 leader networks, named networks, and anonymous networks are defined, and some algorithms are given to simulate one type of network on the other. In Subsection 2.3 a lower bound on the complexity of network orientation is proved. In Subsection 2.4 it is proved that the problem cannot be solved on anonymous networks using deterministic algorithms.

2.1 Definitions of Networks and Orientations

Processor networks are identified with graphs, where the nodes are the processors and an edge between two nodes exists if the corresponding processors are connected by a communication channel. For each topology a labeling will be defined as an assignment, in each node, of labels to the edges of each node. An orientation is defined as a labeling where the labels satisfy an additional global consistency property. A different characterization of orientations, based on the sequence of labels found on the edges of any closed path, is given in Section 6; that section is very technical.

The Clique. The *clique* (on N nodes) is a network (consisting of N nodes), where each node is connected to every other node. The clique on N nodes has $\frac{1}{2}N(N-1)$ edges, and every node has degree $N-1$. A *labeling* of the clique is an assignment in every node of different labels from the set $\{1,..,N-1\}$ to the edges incident to that node. An *orientation* of the clique is a labeling, for which each node v can be assigned a unique name $\mathcal{N}(v)$ from the set $\{0,1,..,N\}$, such that the edge connecting node v to node w is labeled $(\mathcal{N}(w)-\mathcal{N}(v) \bmod N)$ in node v.

The Hypercube. The n–dimensional *hypercube* ($n>0$) is a network consisting of $N=2^n$ nodes, where each node can be assigned a unique name from the set $\{(b_0,...,b_{n-1}) : b_i = 0,1\}$, in such a way that node $b=(b_0,...,b_{n-1})$ is connected to the nodes $(b_0,...,\bar{b_i},...,b_{n-1})$, $i=0...n-1$. The n–dimensional hypercube has $\frac{1}{2}nN$ edges, and every node has degree n. A *labeling* of the hypercube is an assignment in every node of different labels from the set $\{0,1,..,n-1\}$ to the edges incident to that node. An *orientation* of the hypercube is a labeling, for which each node v can be assigned a unique name $\mathcal{N}(v)=(b_0,...,b_{n-1})$, in such a way that the edge connecting nodes v and w is labeled i in both v and w if $\mathcal{N}(v)$ and $\mathcal{N}(w)$ differ in bit i.

The Torus. The $n \times n$ *torus* is a network consisting of $N=n^2$ nodes, where each node can be assigned a unique name from the set $\mathbb{Z}_n \times \mathbb{Z}_n$, in such a way that node (i,j) is connected to the four nodes $(i,j+1)$, $(i,j-1)$, $(i+1,j)$, and $(i-1,j)$. (Addition and subtraction here is $\bmod n$.) The $n \times n$ torus has $2n^2$ edges, and every node has degree 4. A *labeling* of the torus is an assignment in every node of different labels from the set $\{up,\ down,\ right,\ left\}$ to the edges incident to that node. An *orientation* of the torus is

a labeling, for which each node v can be assigned a unique name $\mathcal{N}(v) = (i, j)$, such that the edge (v, w) is labeled *up* (or *down*, *right*, *left*, respectively) in v if $\mathcal{N}(w) = (i, j + 1)$ (or $(i, j - 1)$, $(i + 1, j)$, $(i - 1, j)$, respectively).

For a labeling $\mathcal{L}$, let $\mathcal{L}_u(v)$ be the label assigned to edge (u, v) in node u. A network is said to be *labeled* if a labeling is known to the processors in that network, and *oriented* if an orientation is known. To allow a processor to distinguish between its links, it is assumed that a labeling $\mathcal{L}$ of the network is given initially. The aim of an orientation algorithm (for cliques, hypercubes or tori, respectively) is to terminate in each node v with a *permutation* π_v (of $\{1, ..., N-1\}$, $\{0, ..., n-1\}$, or $\{up, down, right, left\}$, respectively), such that the labeling $\mathcal{O} = \pi(\mathcal{L})$, defined by $\mathcal{O}_v(w) = \pi_v(\mathcal{L}_v(w))$, is an orientation.

2.2 Network Models

In this paper it is assumed that processors and communication are *asynchronous* and *reliable*. The time between two steps of one processor may be arbitrarily large (but is always finite) and messages that are sent will be received after an arbitrarily large, but finite delay, and unaltered. It is not assumed that messages, sent over the same link, will be received in the order in which they were sent. The number of processors is denoted by N and the number of links by E. The complexity of a distributed algorithm is expressed as the number of messages exchanged in an execution of the algorithm. A more precise measure is the *bit complexity*, which is expressed as the total number of bits transmitted in the messages together.

A distributed algorithm consists of a local algorithm for each processor. Different assumptions about the required *symmetry* of the algorithm are considered.

Leader Network. A network is called a *leader network* or said to *contain a leader* if there is exactly one processor which knows that it is "the leader". The availability of a leader can be exploited by providing a distinguished local algorithm for it, while all other processors execute the same local algorithm (which is different from the leader algorithm).

Named Network. A network is called a *named network* if each processor is assigned a unique *name* (an identification number). The name of a processor is known to that processor, but not to other processors.

Besides uniqueness no assumptions about the names may be used to prove the correctness of the algorithm (such as, that the numbers are taken from a certain bounded range $\{1, ..., M\}$). For the analysis of the bit complexity of algorithms, however, it is usually assumed that a name is represented in $O(\log N)$ bits.

Anonymous Network. A network is called an *anonymous network* if all processors execute the same algorithm and no names are known.

The Power of Leader and Named Networks. It has turned out that leader networks and named networks are equivalent in terms of the computations that can be carried out on these networks. Each of the two models can be simulated by the other, because in a leader networks unique names can be assigned, and in named networks a leader can be elected. Algorithms for this purpose will be given in Subsection 2.2.1.

Deterministic and Randomized Algorithms. In order to express the computational power of anonymous networs, as compared to leader or named networks, it is necessary to distinguish between *deterministic* and *randomized* algorithms. The execution model of asynchronous distributed systems is a *non–deterministic* one. This means that the next step in a computation is in general not uniquely defined by the (global) state of a computation. As an example one may consider the situation where two processors have sent a message to a third processor v. As v is usually able to receive a message from any of its links, the next step in the computation is chosen by the run–time system, which defines which of the two messages will be received first. Thus in general a distributed algorithm (even if v's reaction to the receipt of a message is specified precisely and deterministically) describes a *class of possible executions* rather than a single execution.

An algorithm is deterministic if the processors terminate in *each* possible execution of this class. For a randomized algorithm it is not required that the processors terminate in *each* execution of the algorithm, but only that this happens with a high probability (1 usually). (A probability distribution on the class of executions must be defined.) Thus, although infinite executions of a randomized algorithm may exist (and usually do exist), the algorithm is nonetheless regarded correct if the probability of such an execution is 0.

Note that an algorithm being a deterministic algorithm does not imply that its output is completely determined by its input; the non–determinism of the execution model may result in a large number of different executions, each with different outcome. For example, the Echo algorithm (to be described later; see Algorithm 1) is a deterministic algorithm, but every spanning tree of the network is a possible outcome of the algorithm. The determinism of the algorithm refers to the fact that each of the possible executions terminates, not that there exists only one execution or that all executions yield the same result.

The Power of Anonymous Networks. Anonymous networks are weaker than leader and named networks in terms of the computations that they can perform. Leader and named networks can simulate anonymous networks (namely, by making the leader algorithm equal to the non–leader algorithm, or by not using the name, respectively). On the other hand, Angluin [Ang80] has shown that no deterministic algorithms exists to elect a leader in anonymous networks. With arguments similar to hers it will be shown that no deterministic algorithms for orienting anonymous networks exist. As deterministic orientation algorithms do exist for leader and named networks, it follows that anonymous networks can compute strictly less functions than leader or named functions when deterministic algorithms are used. Consequently, randomized algorithms must be used for election, orientation, and other tasks in anonymous networks.

A leader can be elected in an anonymous network by a randomized algorithm, provided that N is known to the processors (Theorem 2.11). This implies that a leader can be elected in anonymous cliques and hypercubes, because the size of those networks N can be computed from the degree of a node. The algorithm will be presented in Subsection 2.2.2. It was shown by Itai and Rodeh [IR81] that there exists no randomized election algorithm (for rings) when the number of nodes is unknown[1]. With an argument similar to theirs it will be shown (see Subsection 5.3) that there exists no randomized algorithm to orient anonymous tori of unknown size. As orientation algorithms do exist for leader and named

[1] As a technical detail it should be noted here that according to our definitions only *processor terminating* algorithms are considered; see also Subsection 7.3.

tori of unknown size (see Section 5), it follows that anonymous networks can compute strictly less functions than leader or named networks, even when randomization is allowed.

2.2.1 Leader Networks and Named Networks

In this subsection an algorithm is described to elect a single processor as a leader in a named network, which uses (at most) $O(EN)$ messages. Subsequently an algorithm is described to assign unique names in a leader network, which uses $2E + N - 1$ messages. These algorithms prove the following two theorems.

Theorem 2.1 *If a problem can be solved using M messages on a leader network, it can be solved using $M + O(NE)$ messages on a named network.*

Theorem 2.2 *If a problem can be solved using M messages on a named network, it can be solved using $M + 2E + N - 1$ messages on a leader network.*

The problem of electing a leader (in a named network) has received considerable attention during the last decade, and more efficient solutions than the one described here are known. The Spanning Tree algorithm proposed by Gallager, Humblet, and Spira [GHS83] can be used to elect a leader using $O(N \log N + E)$ messages, which implies the following, stronger result.

Theorem 2.3 *If a problem can be solved using M messages on a leader network, it can be solved using $M + O(N \log N + E)$ messages on a named network.*

Electing a Leader in a Named Network. The algorithm to elect a leader in a named network uses as a building block an algorithm known as the *Echo algorithm*. Using the Echo algorithm (Algorithm 1) a single processor can flood its name over the network and eventually receive a confirmation that *all* processors have received its identity.

The flooding of the processor's name is initiated by sending a message to all neighbors. Processors receiving the name for the first time forward it, and record the link on which they first received the name, thus defining a spanning tree in the network. A processor confirms that all processors in its subtree have received the name by "echoing" the name to its father in the tree. The initiator terminates after receipt of a message (either an echo or a flooding message) from all of its neighbors. When this happens, all processors have confirmed the receipt of the initiator's identity (as proved, for example, in [Tel94, Sec. 6.2.3]).

The variables for each processor v are: $name_v$ and dgr_v, the name of v and number of links of v (constants for v actually); $rcvd_v$, the number of messages that v has received; and $father_v$, the link over which v first received a message. The name n of the initiator is transmitted in a $\langle \mathbf{name}, n \rangle$ message.

The algorithm for leader election (Algorithm 2) is obtained from the Echo algorithm through the application of a mechanism called *extinction* (cf. [Tel94, Sec. 7.3.1]); see also below. To elect a leader, each processor initiates the flooding of its own identity using the Echo algorithm. However, processor v processes $\langle \mathbf{name}, n \rangle$ messages only if $n \geq name_v$. Moreover, if v has ever received a $\langle \mathbf{name}, n \rangle$ message (with $n \geq name_v$), it processes $\langle \mathbf{name}, n' \rangle$ messages only if $n' \geq n$. To this end, processor v maintains a variable $larnm_v$, which is the largest name n that v has seen.

```
Processor v is the initiator:
    begin rcvd_v := 0 ;
        for l = 1 to dgr_v do send ⟨ name, name_v ⟩ via link l ;
        while rcvd_v < dgr_v do
            begin receive msg via link l ;
                rcvd_v := rcvd_v + 1
            end
    end .

Processor v is not the initiator:
    begin rcvd_v := 0 ;
        while rcvd_v < dgr_v do
            begin receive msg via link l ;
                if rcvd_v = 0 then
                    (* First message defines the spanning tree *)
                    begin father_v := l ;
                        forall links k ≠ l do send msg via k
                    end ;
                rcvd_v := rcvd_v + 1
            end ;
        send msg through link father_v
    end .
```

Algorithm 1: THE ECHO ALGORITHM.

As a result, only the flood initiated by the processor with the largest name (w say) is processed by all other processors, and thus only processor w terminates the Echo algorithm. When this happens, w floods $\langle \textbf{lead}, name_w \rangle$ messages to all processors to inform them about the leader. The processors terminate the election when they have received a $\langle \textbf{lead}, n \rangle$ message via every link. To this end, processor v maintains a variable $ldrc_v$ to count the number of $\langle \textbf{lead}, n \rangle$ messages it has received. Upon termination v considers itself leader iff $leader_v = name_v$. The processor with the largest name, and only this processor, considers itself leader.

The properties of the algorithm are summarized in the following theorem.

Theorem 2.4 *There exists a deterministic algorithm to elect a leader in a named network. The algorithm exchanges $O(EN)$ messages.*

Assigning Names in a Leader Network. The algorithm to assign names in a leader network, Algorithm 3/4, consists of two global phases, each initiated by the leader. The first phase, which again relies on the Echo algorithm, constructs a spanning tree in the network and computes, for each node, the size of the subtree of each of its children. In its echo, processor v reports the size of its subtree. In the second phase the leader assigns itself the number 0, and distributes the remainder of the set $\{0, .., N-1\}$ over its children, where each child receives as many numbers as there are nodes in its subtree. Each node, upon receipt of an interval of numbers from its father, assigns itself the smallest number and distributes the remainder of the interval over its children in a similar manner.

```
begin rcvd_v := 0 ; larnm_v := name_v ; ldrc_v := 0 ;
      for l = 1 to dgr_v do send ⟨ name, name_v ⟩ via link l ;
      while ldrc_v < dgr_v do
         begin receive message msg via link l ;
               if msg = ⟨ lead, n ⟩ then
                  begin if ldrc_v = 0 then
                              (* First ⟨ lead, n ⟩ message, forward *)
                              forall k = 1..dgr_v do send ⟨ lead, n ⟩ via k ;
                        ldrc_v := ldrc_v + 1 ; leader_v := n
                  end
               else (* a ⟨ name, n ⟩ message *)
                  begin
                    if n > larnm_v then
                      (* Larger name, this implies first receipt! *)
                      begin larnm_v := n ; rcvd_v := 0 ; father_v := l ;
                            forall k ≠ l do send ⟨ name, n ⟩ via k
                      end ;
                    if n ≥ larnm_v then
                      begin rcvd_v := rcvd_v + 1 ;
                            if rcvd_v = dgr_v then
                              if n = name_v
                                then forall k do send ⟨ lead, name_v ⟩ via k
                                else send ⟨ name, n ⟩ via father_v
                      end
                    (* If n < larnm_v the message is ignored. *)
                  end
         end
      (* Processor v is the leader iff leader_v = name_v *)
end .
```

Algorithm 2: LEADER ELECTION WITH THE ECHO ALGORITHM.

The algorithm uses three types of messages, namely $\langle$ **forw** $\rangle$, $\langle$ **son**, s $\rangle$, and $\langle$ **inte**, a, b $\rangle$ messages. The variables of processor v are: $rcvd_v$, dgr_v, and $father_v$ as in the Echo algorithm; $subtr_v[1..dgr_v]$, the size of the subtree of each child; and $name_v$, the name that v will be assigned by the algorithm. The algorithm terminates in each processor, and when processor v terminates, $name_v$ is an integer in $\{0, .., N-1\}$, different from $name_w$ for every $w \neq v$.

The properties of the naming algorithm are summarized in the following theorem.

Theorem 2.5 *There exists a deterministic algorithm to assign names in a leader network. The algorithm exchanges exactly $2E + N - 1$ messages.*

It is interesting to note, that the processors do not start phase 2 simultaneously, but rather each processor does so at its own time. A similar algorithm, which sends echo's also in the second phase and therefore has a message complexity of $2E + 2(N-1)$ was given by Bouabdallah and Naimi [BN89]. The echo's of the second phase make the leader the last process to terminate, so that termination of the leader signals termination of the entire algorithm.

```
Processor v is the leader:
    begin rcvd_v := 0 ;
        for l = 1 to dgr_v do send ⟨ forw ⟩ via link l ;
        while rcvd_v < dgr_v do
            begin receive msg via link l ; rcvd_v := rcvd_v + 1 ;
                if msg = ⟨ son, s ⟩ then subtr_v[l] := s
                                    else subtr_v[l] := 0
            end ;
        (* Start phase 2 *)
        name_v := 0 ; given := 0 ;
        for l = 1 to dgr_v do
            if subtr_v[l] > 0 then
                begin send ⟨ inte, given + 1, given + subtr_v[l] ⟩ via l ;
                      given := given + subtr_v[l]
                end
        (* given now equals the number of processors *)
    end .
```

Algorithm 3: ASSIGNING NAMES IN A LEADER NETWORK (LEADER).

Extinction. To apply an algorithm for a leader network to a named network it is not necessary to pass through a separate election phase as described in Algorithm 2 or [GHS83]. It is possible to combine the election with an algorithm for a leader network by applying the extinction principle to this algorithm directly. To be more precise about this construction, let *LNA* be an algorithm that solves some network problem for a leader network. (*LNA* stands for "leader network algorithm".) The following two assumptions are made about *LNA*.

1. *LNA* is initiated only by the leader.
2. All processors are involved in every possible execution of *LNA* and have terminated before the leader terminates.

These assumptions hold for most algorithms for leader networks, but are not implied by the definition of a leader network. Every leader algorithm, however, can be modified (in a straight–forward way and at the expense of at most $2E$ extra messages) so as to satisfy these assumptions.

An algorithm *NNA* for a named network is constructed as follows. (*NNA* stands for "named network algorithm".) Each processor v has all the variables of algorithm *LNA* (those of the leader as well as those of the non–leaders), and a variable $larnm_v$, which is initialized to $name_v$. Each processor initiates algorithm *LNA* (as if it were the leader), but tags all messages with its name. When a message of *LNA* is received, the name n contained in it is compared with $larnm_v$. If $n < larnm_v$ the message is simply ignored. If $n = larnm_v$ the message is processed as in algorithm *LNA* (the leader part if $n = name_v$, the non–leader part if $n > name_v$). If $n > larnm_v$, v resets the variables of *LNA* to their initial value, sets $larnm_v := n$, and processes the message as in (the non–leader part of) *LNA*.

```
Processor v is not the leader:
    begin rcvd_v := 0 ;
        while rcvd_v < dgr_v do
          begin receive msg via link l ;
                if rcvd_v = 0 then
                  (* The first message defines the spanning tree *)
                  begin father_v := l ;
                        forall links k ≠ l do send ⟨forw⟩ via k
                  end ;
                rcvd_v := rcvd_v + 1 ;
                if msg = ⟨son, s⟩ then subtr_v[l] := s
                                  else subtr_v[l] := 0
          end ;
        (* Report to the father in the tree *)
        size := 1 + Σ_{l=1}^{dgr_v} subtr_v[l] ;
        send ⟨son, size⟩ via link father_v ;
        (* Phase 2 *)
        receive ⟨inte, a, b⟩ ;
        name_v := a ; given := a ;
        for l = 1 to dgr_v do
            if subtr_v[l] > 0 then
              begin send ⟨inte, given + 1, given + subtr_v[l]⟩ via l ;
                    given := given + subtr_v[l]
              end
    end .
```

Algorithm 4: ASSIGNING NAMES IN A LEADER NETWORK (NON–LEADER).

Let w be the processor for which $name_w > name_v$ for all $v \neq w$. No processor $v \neq w$ succeeds to terminate the execution of *LNA* it initiated, as w does not cooperate in this execution (use assumption 2). Eventually, all processors cooperate the execution of *LNA* which was initiated by w. When this execution terminates, the network problem is solved by this execution of *LNA*. The construction of algorithm *NNA* proves the following result.

Theorem 2.6 *If a problem can be solved using M messages on a leader network, it can be solved using $O(NM)$ messages in the worst case on a named network.*

Regardless of the function computed by *LNA*, algorithm *NNA* implicitly performs an election, because exaclty one processor (w) succeeds to terminate its own execution of *LNA*. The number of messages sent by algorithm *NNA* is usually much higher than the number stated in Theorem 2.3. The extinction construction compares more favorable with the earlier result when the time complexity is considered. The election algorithm referred to in Theorem 2.3 uses time proportional to N (in the worst case), so that the separate election stage adds $\Omega(N)$ time to the time needed by algorithm *LNA*. The extinction construction results in an algorithm that runs in the same amount of time as the original algorithm (when time is measured from the moment that all processors have started).

Furthermore, in several particular cases it has been observed that the *average case* complexity of the resulting algorithm is much better than its worst case complexity. Chang

and Roberts [CR79] proposed an election algorithm where extinction is applied to an algorithm in which the leader sends a message on a *ring* of processors and receives it back after N steps. They proved that the worst case complexity of their algorithm is $O(N^2)$, and that the average case complexity is $O(N \log N)$. Mattern [Mat89] has shown that the average case complexity of Algorithm 2 is $O(E \log N)$. These results suggest the following (open) question.

Open Question 2.7 *What conditions must be satisfied by algorithm LNA to guarantee that the average case complexity of algorithm NNA equals* $\log N$ *times the complexity of algorithm LNA?*

2.2.2 Randomized Algorithms for Anonymous Networks

In this subsection an algorithm (based on an algorithm by Matias and Afek [MA89]) is presented to elect a leader in an anonymous network of which the number of processors is known to each processor. The algorithm operates in phases, each of which is very similar to the election algorithm for named networks.

Each processor starts as a candidate in phase 1; see Algorithm 5. To start a phase, each candidate selects a name using a random function, and initiates the flooding of this name using the Echo algorithm. Due to the possibility that different processors select the same name, a processor may terminate the Echo algorithm as the root of a tree which does not cover all processors. To detect this situation, the echos report the number of processors in each subtree (as in Algorithm 3/4). When a processor terminates the Echo algorithm as the root of a tree of N processors, it becomes the leader. When the number of processors in the tree is smaller than N, the processor proceeds to the next phase as a candidate.

To allow for a more compact coding of the algorithm, the phase number is made part of the name of a processor. Crucial for the correctness of this algorithm is, that whenever a processor has sent an echo ($\langle \mathbf{sns}, n, s \rangle$) it will thereafter never be a candidate and never be elected. This is because after the sending of such a message by v, $larnm_v > name_v$ continues to hold, so v will never process a message carrying $name_v$.

Lemma 2.8 *If a processor starts phase k, no processor is elected in phase $k' < k$.*

Proof. Assume processor w starts phase k, then processor w has never sent and will never send a $\langle \mathbf{sns}, n, s \rangle$ message for a phase $k' < k$. To become elected in phase k', processor v must become the root of a tree of N nodes, all except the root having send a $\langle \mathbf{sns}, name_v, s \rangle$ message. □

Lemma 2.9 *If processor v becomes elected in phase k, $name_v$ in phase k is larger than all other names selected in phase k.*

Proof. To become elected, processor v first becomes the root of a tree of N processors, which implies that all processors have sent a $\langle \mathbf{sns}, name_v, s \rangle$ message. The sending of such a message by processor u implies $name_u < larnm_u$, which continues to hold thereafter. □

Lemma 2.10 *Algorithm 5 terminates with probability 1 in all processors, and when processor v terminates $leader_v = name_w$, the largest name of any processor. This name is the name of exactly one processor.*

```
Procedure Newphase:
    begin phase_v := phase_v + 1 ; tmp := rand ;
          name_v := (phase_v, tmp) ; rcvd_v := 0 ; larnm_v := name_v ;
          for l = 1 to dgr_v do send ⟨ name, name_v ⟩ via link l
    end
begin
  phase_v := 0 ; ldrc_v := 0 ; Newphase ;
  while ldrc_v < dgr_v do
    begin
      receive message msg via link l ;
      if msg = ⟨ lead, n ⟩ then
        begin if ldrc_v = 0 then
                  forall k = 1..dgr_v do send ⟨ lead, n ⟩ via k
              ldrc_v := ldrc_v + 1 ; leader_v := n ;
        end
      else (* a ⟨ name, n ⟩ or ⟨ sns, n, s ⟩ message *)
        begin
          if n > larnm_v then
            (* Larger name, this implies first receipt.
               Processor v is defeated forever. *)
            begin larnm_v := n ; rcvd_v := 0 ;
                  father_v := l ; subtr_v[l] := 0;
                  forall k ≠ l do send ⟨ name, n ⟩ via k
            end ;
          if n ≥ larnm_v then
            begin rcvd_v := rcvd_v + 1 ;
                  if msg = ⟨ sns, n, s ⟩
                    then subtr_v[l] := s else subtr_v[l] := 0 ;
                  if rcvd_v = dgr_v then
                    if n = name_v then
                      if 1 + Σ_{k=1}^{dgr_v} subtr_v[l] = N
                        then forall k do send ⟨ lead, name_v ⟩ via k
                        else Newphase
                      else send ⟨ sns, n, 1 + Σ_{k=1}^{dgr_v} subtr_v[l] ⟩ via father_v
            end
        end
    end
end .
```

Algorithm 5: ELECTION FOR ANONYMOUS NETWORKS.

Proof. Consider a phase k that is started by one or more processors. Assuming that no processor starts phase $k+1$, the processor(s) with largest name in phase k will receive all the echo's necessary to pass through the Echo algorithm in phase k. Hence, if no processor starts phase $k+1$ and there is a *single* processor with the largest name it will be elected (because it is root of a tree of size N). Moreover, if no processor starts phase $k+1$ and there are *multiple* processors with the largest name in phase k they will start phase $k+1$ (because they are root of a tree of size smaller than N), which is a contradiction. Thus,

once phase k is started, either a processor becomes elected in that phase, or phase $k+1$ will be started.

There is a positive constant ρ (depending on the probability distribution of the *rand* function) such that, if more than one processor starts phase k, at least one processor will become defeated in phase k with probability at least ρ. This implies that with probability 1 eventually all processors except one become defeated.

The remaining processor w becomes elected and floods $\langle \textbf{lead}, name_w \rangle$ messages over the network, which cause all processors v to terminate with $leader_v = name_w$. □

Theorem 2.11 *There exists a randomized algorithm for election in anonymous networks of known size, which terminates with probability 1.*

The expected message complexity depends on the probability distribution of the *rand* function. It is left as an open problem to the reader to obtain a complexity as low as possible.

2.3 Lower Bounds for Network Orientation

Let N denote the number of processors and E the number of links in the network. In this subsection a lower bound of $\Omega(E)$ messages is shown on the message complexity of orientation algorithms, for the topologies considered in this paper.

Theorem 2.12 *Any algorithm for the orientation of cliques, hypercubes or tori exchanges at least $E - \frac{1}{2}N$ messages in every execution.*

Proof. For a labeling $\mathcal{L}$, let $\mathcal{L}^{u,v,w}$ (where v and w are neighbors of u) be the labeling defined by $\mathcal{L}^{u,v,w}_u(v) = \mathcal{L}_u(w)$, $\mathcal{L}^{u,v,w}_u(w) = \mathcal{L}_u(v)$, and all other labels of $\mathcal{L}^{u,v,w}$ are as in $\mathcal{L}$. ($\mathcal{L}^{u,v,w}$ is obtained by exchanging $\mathcal{L}_u(w)$ and $\mathcal{L}_u(v)$.) It can be verified that for every orientation $\mathcal{O}$ (of a clique, hypercube or torus), and every v, u, and w, $\mathcal{O}^{u,v,w}$ is *not* an orientation.

Consider an execution of an orientation algorithm, with initial labeling $\mathcal{L}$, that terminates with a permutation π_v for each node (where $\mathcal{O} = \pi(\mathcal{L})$ is an orientation). Assume furthermore that in this execution some node u did not send nor receive any message to or from its two neighbors v and w. As u has not communicated with v, nor with w, the same execution is possible if the network is initially labeled with $\mathcal{L}^{u,v,w}$, and all processors terminate with the same permutation. However, $\mathcal{O}' = \pi(\mathcal{L}^{u,v,w}) = \mathcal{O}^{u,v,w}$ is *not* an orientation, and the algorithm is not correct.

It follows, that in every execution every node must communicate with at least all its neighbors except one. □

Corollary 2.13 *The orientation of the N clique requires the exchange of $\Omega(N^2)$ messages. The orientation of the n-dimensional hypercube requires the exchange of $\Omega(n2^n)$ messages. The orientation of the $n \times n$ torus requires the exchange of $\Omega(n^2)$ messages.*

Along the same line a lower bound on the bit (and/or time) complexity can be derived. A generalization of the argument used in the proof of Theorem 2.12 derives an incorrect result if the communication patterns on (u, v) and (u, w) are identical (but not necessarily

empty in both cases). This means that the same messages are exchanged on (u, v) and (u, w) (and at the same time, if time is considered). The argument then shows that for each u there are at least $\delta(u)$ different patterns of communication. ($\delta(u)$ denotes the degree of u, and as the considered networks are regular, we henceforth write δ instead.) If only one message is sent per link (and no information is coded in the *time* at which it is sent), it contains at least $\log \delta$ bits, and the bit complexity is $\Omega(E \log \delta)$. When the time of sending a message is used to code information, a lower bit complexity may be achievable.

2.4 Deterministic Orientation of Anonymous Networks

In this subsection it will be shown that the orientation problem cannot be solved for anonymous networks by using deterministic algorithms. This result is obtained by providing symmetric initial labelings that cannot be turned into an orientation by a deterministic algorithm.

Definition 2.14 *A labeling $\mathcal{L}$ (of a clique, hypercube or torus) is* symmetric *if there exists a permutation σ (of $\{1, .., N-1\}$, $\{0, .., n-1\}$ or $\{up, down, right, left\}$) such that for all links (u, v), $\mathcal{L}_u(v) = \sigma(\mathcal{L}_v(u))$.*

Equivalently, $\mathcal{L}$ is symmetric if for all nodes u, v, $, u'$, v', $\mathcal{L}_u(v) = \mathcal{L}_{u'}(v')$ implies $\mathcal{L}_v(u) = \mathcal{L}_{v'}(u')$. A labeling is a pre–orientation if it can be turned into an orientation by application of the same permutation in every node.

Definition 2.15 *A labeling $\mathcal{L}$ is a* pre–orientation *if there exists a single permutation π_0 such that with $\pi_v = \pi_0$ for all v, $\pi(\mathcal{L})$ is an orientation.*

Theorem 2.16 *If, for an anonymous network, there exists a symmetric labeling which is not a pre–orientation, then this network cannot be oriented by a deterministic algorithm.*

Proof. Let $\mathcal{L}$ be a symmetric labeling, and assume a deterministic algorithm is started in a network with initial labeling $\mathcal{L}$. There is an execution E of this algorithm in which every processor executes exactly the same sequence of steps.

To see this, consider the environment of a processor consisting of its (local) state and the contents of its incoming links. Initially all processors have an identical environment (namely, where the state is the initial one and all links are empty). If at any moment the processors are in identical environments and a step is enabled in one processor, the same step is enabled in all processors. Assume this step is executed in all processors. After the step all processors are again in the same state. If the step included the receipt of a message *msg* via link l, this message is removed from the incoming link l of every processor. If the step included the sending of a message *msg* via link l, this message is added to the incoming link $\sigma(l)$ of every processor, because $\mathcal{L}$ is symmetric. Thus, after the execution of the step in every processor, the processors are in identical environments again.

By assumption, the algorithm terminates. Because every processor has executed the same sequence of steps, it terminates with the same permutation π_0 in every processor. Because $\mathcal{L}$ is not a pre–orientation, $\pi_0(\mathcal{L})$ is not an orientation, and the algorithm is incorrect. □

It remains to show that there exist symmetric labelings that are not pre–orientations.

Corollary 2.17 *There exists no deterministic algorithm to orient an anonymous clique of size N, where N is not prime.*

Proof. It is left as an exercise to find a suitable labeling for $N = 4$. For $N > 4$, write $N = pq$ with $1 < p < N$ and $q \neq 2$. For convenience, number the nodes from 0 to $N-1$, and define $\mathcal{L}$ as follows. Set $\mathcal{L}_v(u) = (u - v) \bmod N$ when p is not a divisor of u or not a divisor of v. When p divides both u and v, set $\mathcal{L}_v(u) = (v - u) \bmod N$.

To show that $\mathcal{L}$ is symmetric, observe that $\mathcal{L}_u(v) + \mathcal{L}_v(u) = N$ always (so $\sigma(l) = N - l$ satisfies the requirement in Definition 2.14).

To show that $\mathcal{L}$ is not a pre–orientation, observe first that for an orientation $\mathcal{O}$ of the clique, the following holds for every node v and all labels a and b. If, starting from node v, first the link labeled with a is traversed, and (from the reached node) the link labeled with b, the same node is reached as when first b is traversed and then a. As permuting the labels of all links does not change this property, the same holds for pre–orientations.

Labeling $\mathcal{L}$, however, does not satisfy this property, as is easily verified with $v = 0$, $a = 1$, $b = p$. Starting from node 0, traversing the link labeled with 1 and then the link labeled with p, one arrives in node $p+1$. Traversing first the link labeled with p and then the link labeled with 1, one arrives in node $N - p + 1$. Finally, $p + 1 \neq N - p + 1$ because $q \neq 2$. □

Suitable labelings are easily found for hypercubes and tori of even size because their bipartiteness can be employed.

Corollary 2.18 *There exists no deterministic algorithm to orient an anonymous hypercube of dimension $n > 1$.*

Proof. For convenience, label the nodes with bitstrings of length n as in the definition of the hypercube. Call a node *even* if it has an even number of 1's in its bitstring and *odd* if it has an odd number of 1's. For v labeled with $(b_0, ..., b_{n-1})$ and u labeled with $(b_0, ..., \bar{b_i}, ..., b_{n-1})$, set $\mathcal{L}_u(v) = i$ when u is even, and $\mathcal{L}_u(v) = n - 1 - i$ when u is odd.

To show that $\mathcal{L}$ is symmetric, observe that $\mathcal{L}_u(v) + \mathcal{L}_v(u) = n - 1$ always (because every link connects an even node with an odd node).

To show that $\mathcal{L}$ is not a pre–orientation, observe that for an orientation $\mathcal{O}$ of the hypercube, for every u and v, $\mathcal{O}_u(v) = \mathcal{O}_v(u)$. As permuting the labels of all links does not change this property, the same holds for pre–orientations.

Labeling $\mathcal{L}$, however, does not satisfy this property (for $n > 1$), as is easily verified. □

Corollary 2.19 *There exists no deterministic algorithm to orient an anonymous torus of even size.*

Proof. For convenience, label the nodes with elements of $\mathbb{Z}_n \times \mathbb{Z}_n$, as in the definition of the torus. Call the node labeled with (i, j) *even* if $2|(i + j)$, and *odd* otherwise. For node v labeled with (i, j) and u labeled with $(i, j + 1)$ ($(i, j - 1)$, $(i + 1, j)$, $(i - 1, j)$), let $\mathcal{L}_v(u) = up$ (*down*, *right*, *left*) if v is even, and $\mathcal{L}_v(u) = down$ (*up*, *left*, *right*) if v is odd.

To show that $\mathcal{L}$ is symmetric, observe that $\mathcal{L}_u(v) = \mathcal{L}_v(u)$ for every link (u, v) (as one of them is even and the other is odd).

To show that $\mathcal{L}$ is not a pre–orientation, observe that for an orientation $\mathcal{O}$ of the torus, for every u and v, $\mathcal{O}_u(v) \neq \mathcal{O}_v(u)$. As permuting the labels of all links does not change this

```
begin for l = 1 to N − 1 do send ⟨ name, name_v ⟩ via l ;
      rcvd_v := 0 ;
      while rcvd_v < N − 1 do
         begin receive ⟨ name, n ⟩ via link l ;
               rcvd_v := rcvd_v + 1 ; neigh_v[l] := n
         end ;
      (* Compute node label *)
      label_v := #{k : neigh_v[k] < name_v} ;
      for l = 1 to N − 1 do
         begin (* Compute neighbor's node label and link label *)
               ll := #{k : neigh_v[k] < neigh_v[l]} ;
               if name_v < neigh_v[l] then ll := ll + 1 ;
               π_v[l] := (ll − label_v) mod N
         end
end .
```

Algorithm 6: ORIENTATION OF NAMED CLIQUES.

property, the same holds for pre–orientations. Labeling $\mathcal{L}$ does not satisfy this property as observed above. □

The problem of deterministically orienting anonymous tori was addressed from a different point of view by Syrotiuk *et al.* [SCP93]. Syrotiuk *et al.* studied the question of restricting the permissible initial permutations such that a deterministic solution becomes possible.

3 The Orientation of Cliques

In this section algorithms for the orientation of cliques will be given. All algorithms described in this section determine an assignment of unique labels from the set $\{0, .., N-1\}$ to the nodes and obtain the orientation from this node labeling.

3.1 Named Cliques

In case the network is named, the label of processor v will be the number of processors with a smaller name (its *rank*; see Algorithm 6). The names are exchanged using $\langle$ **name**, n $\rangle$ messages, and stored in an array $neigh_v[1..N-1]$ in processor v.

Theorem 3.1 *There exists a deterministic algorithm to orient named cliques, which uses* $N(N-1)$ *messages (of* $O(\log N)$ *bits each) and completes in time* $O(1)$.

The computation of all ranks in the second part of the algorithm is done more efficiently by sorting the names, but this still costs $\Omega(N \log N)$ internal processing time. An algorithm requiring less internal processing is obtained, when a processor only computes its own rank locally, after which the ranks are exchanged in a second round of communication.

3.2 Leader Cliques

Processor v is the leader:

begin $label_v := 0$;
  for $l = 1$ **to** $N - 1$ **do**
    begin send $\langle$ **youare**, $l \rangle$ via link l ;
      $\pi_v[l] := l$
    end
end .

Processor v is not the leader:

begin receive $\langle$ **youare**, $num \rangle$ via link l ;
  $label_v := num$; $\pi_v[l] := N - num$; $rcvd_v := 1$;
  forall $k \neq l$ **do** send $\langle$ **iam**, $num \rangle$ via k ;
  while $rcvd_v < N - 1$ **do**
    begin receive $\langle$ **iam**, $num \rangle$ via link l ;
      $rcvd_v := rcvd_v + 1$; $\pi_v[l] := (num - label_v) \bmod N$
    end
end .

Algorithm 7: ORIENTATION OF A LEADER CLIQUE.

If a leader w is available, w enforces its local link labeling upon the whole network by assigning processor v its label of the link connecting w to v; see Algorithm 7. The leader assigns the labels using $\langle$ **youare**, $l \rangle$ messages, and the other nodes communicate the names among them using $\langle$ **iam**, $l \rangle$ messages. A non–leader processor must receive a $\langle$ **youare**, $l \rangle$ message as the first message in this algorithm. Earlier arriving $\langle$ **iam**, $l \rangle$ messages are supposed to be implicitly buffered and processed after the receipt of the $\langle$ **youare**, $l \rangle$ message.

Theorem 3.2 *There exists a deterministic algorithm to orient leader cliques, which uses* $(N-1)^2$ *messages (of* $O(\log N)$ *bits each) and completes in time* $O(1)$.

3.3 Anonymous Cliques

According to Corollary 2.17, there exists no deterministic algorithm for the orientation of anonymous cliques (when N is not prime). In this subsection a randomized algorithm for this task is presented (see Algorithm 8). It will be shown, that the algorithm terminates with probability 1, that an orientation is computed when the processors terminate, and that its parameter C can be chosen such that the message and bit complexity of the algorithm are optimal with a very high probability.

All processors start phase 1 as active processors with an empty name string. In each phase, each active processor draws a random number from the range $[1..C]$ and sends the draw to all processors. All processors (including active ones) wait until they have received the draw of every active processor. The draw is appended to the name string, and a processor becomes passive if its name string is now unique. Otherwise, it is active again in the next phase. The algorithm terminates when there are no active processors in a phase. By then, every processor has a unique name string, and the processors are ranked according to the (lexicographic) order of the name strings. The link labels are chosen as in the algorithm for named networks.

```
begin for l = 0 to N − 1 do intvl_v[l] := (0, N − 1) ;
      active_v := N ; phase_v := 1 ;
      while active_v > 0 do
        begin rcvd_v := 0 ;
              if intvl_v[0].lo ≠ intvl_v[0].hi then
                (* v is active, draw label *)
                begin draw_v[0] := rand(1..C) ; rcvd_v := rcvd_v + 1 ;
                      for l = 1 to N − 1 do
                          send ⟨ draw, phase_v, draw_v[0] ⟩ via l
                end ;
              while rcvd_v < active_v do
                begin receive ⟨ draw, phase_v, c ⟩ via link l ;
                      (* Messages of a later phase are not received yet *)
                      rcvd_v := rcvd_v + 1 ; draw_v[l] := c
                end ;
              Split–Intervals ;
              active_v := 0 ;
              for l = 0 to N − 1 do
                  if intvl_v[l].lo ≠ intvl_v[l].hi then active_v := active_v + 1 ;
              phase_v := phase_v + 1
        end ;
      for l = 1 to N − 1 do
          π_v[l] := (intvl_v[l].lo − intvl_v[0].lo) mod N
end .
```

Algorithm 8: ORIENTATION OF AN ANONYMOUS CLIQUE.

In Algorithm 8 the construction of the name strings is done implicitly, and the computation of the ranks is done during the construction of these strings. In this way each processor needs only a bounded amount of storage ($O(\log N)$ bits) per link. To this end, processor v maintains a *ranking interval* $intvl_v[l]$ for each link l and for v itself. In each phase the intervals are split according to the draws in that phase. To allow for compact coding of the algorithm, v stores its own draw and ranking interval in the same array (at location 0).

The variables of processor v are: $phase_v$, the phase v is executing; $active_v$, the number of active processors in the current phase according to v; $rcvd_v$, the number of draws already received by v in the current phase; $draw_v$, an array to store the draw of each active processor in the current phase; and $intvl_v$, an array to store an interval (lo, hi) for each processor. The active processors send their draw in each phase in a $\langle$ **draw**$, p, c \rangle$ message, where p is the phase number and c the draw.

The procedure *Split–Intervals* performs the task of computing the new ranking interval for each processor, given the old intervals and the draws. Before execution of this procedure, (1) the intervals that occur are disjoint and their union is $\{0, .., N-1\}$, and (2) when interval (a, b) occurs there are exactly $b - a + 1$ processors that have this interval. The procedure counts for these $b-a+1$ processors the number of times each draw occurs, and, if b_c is the number of processors with a draw $< c$, assigns the interval $(a + b_c, a + b_{c+1} - 1)$ to the processors with draw c (for $c = 1..C$).

It must first be remarked that for each processor v, the interval computed for v is the same in all processors.

Lemma 3.3 *For processors v, u, w, at the end of each phase $intvl_u[\mathcal{L}_u(v)] = intvl_w[\mathcal{L}_w(v)]$, and $intvl_u[\mathcal{L}_u(v)] = intvl_v[0]$.*

Proof. This equation holds at the beginning of phase 1, each processor is active, and for each processor v $active_v = N$.

Assume $intvl_u[\mathcal{L}_u(v)] = intvl_w[\mathcal{L}_w(v)] = intvl_v[0]$ holds at the beginning of a phase, and for each u $active_u$ equals the number of processors v with $intvl_v[0].lo \neq intvl_v[0].hi$. Exactly the processors v with $intvl_v[0].lo \neq intvl_v[0].hi$ send draws in this phase and each processor waits until exactly this number of draw messages have been received. Thus, *Split–Intervals* is called with the same collection of draws in every processor, and after the execution of *Split–Intervals* in every processor $intvl_u[\mathcal{L}_u(v)] = intvl_w[\mathcal{L}_w(v)] = intvl_v[0]$ holds again. Also, $active_u$ is recomputed as the number of processors v with $intvl_v[0].lo \neq intvl_v[0].hi$ at the end of the phase.

Using induction the integrity of the data is shown to be maintained through each round. □

Theorem 3.4 *When Algorithm 8 terminates an orientation is computed.*

Proof. First observe that it terminates after the same round in every processor, because each processor computes the same number of active processors at the end of each round.

After termination, each processor has an interval of size 1, and the intervals for different processors are disjoint. Thus the intervals define a ranking of the processors, from which the correctness of the computed permutations follows. □

The remainder of this subsection is devoted to the complexity analysis of the algorithm. Let R be the stochastic variable defined as the number of rounds needed in an execution of the algorithm. Define Q_d to be the probability that $R > d$.

Lemma 3.5 $Q_d \leq \min(1, \frac{N^2}{C^d})$.

Proof. To determine the number of rounds one may as well assume that also passive processors continue to extend their name string, as this has no influence on the uniqueness of the name strings of active processors. After d rounds each processor has randomly selected a name string from a universe of $U = C^d$ possible name strings. The probability that more phases are necessary equals the probability that, among N random selections from a universe of size U, there are multiple occurrences of the same selection.

If $U = C^d < N^2$, use that Q_d is a probability so $Q_d \leq 1$. For $U \geq N^2$, find

$$\begin{aligned} Q_d &= \Pr(\text{There are multiple occurrences under } N \\ &\qquad \text{random selections from } U.) \\ &= 1 - \Pr(N \text{ random selections from } U \text{ are all unique.}) \\ &= 1 - \left(\frac{U}{U} \times \frac{U-1}{U} \times \ldots \times \frac{U-N+1}{U}\right) \\ &< 1 - \left(\frac{U-N}{U}\right)^N = 1 - \left(1 - \frac{N}{U}\right)^N < \frac{N^2}{U} \quad \text{for } U \geq N^2. \end{aligned}$$

□

Observe that $Q_d \to 0$ when $d \to \infty$, which implies that the algorithm terminates with probability 1. The expected number of rounds is defined as $E(R) = \sum_{d=0}^{\infty} d \times \Pr(\text{R=d})$, which equals $\sum_{d=0}^{\infty} Q_d$.

Lemma 3.6 $E(R) \leq 2\log_C N + 2$.

Proof.

$$\begin{aligned}
\sum_{d=0}^{\infty} Q_d &= \sum_{d=0}^{2\log_C N - 1} Q_d + \sum_{d=2\log_C N}^{\infty} Q_d \\
&\leq \sum_{d=0}^{2\log_C N - 1} 1 + \sum_{d=2\log_C N}^{\infty} \frac{N^2}{C^d} \\
&\leq 2\log_C N + \left(1 + \frac{1}{C} + \frac{1}{C^2} + \ldots\right) \\
&\leq 2\log_C N + \frac{C}{C-1} \leq 2\log_C N + 2.
\end{aligned}$$

□

With $C = N^2$, the expected number of rounds is less than 3, and the probability that more rounds are used is very small. Thus, the expected number of messages (or bits, respectively) is less than $3N^2$ (or $3N^2(\log C)$, respectively). With C a small constant, the expected number of bits is $O(N^2 \log N)$.

Theorem 3.7 *There exists a randomized algorithm to orient anonymous cliques, which terminates with probability 1.*

4 The Orientation of Hypercubes

In this section algorithms for the orientation of hypercubes will be given. Subsection 4.1 presents an algorithm for a leader hypercube, which uses exactly $2E$ messages. It follows from Theorem 2.3 and Corollary 2.13 that a message optimal algorithm is obtained by preceding this algorithm with an efficient election algorithm. In Subsection 4.2 a different solution is analyzed, namely the algorithm obtained when extinction is applied to the algorithm in Subsection 4.1. Subsection 4.3 considers the problem for anonymous hypercubes.

4.1 Leader Hypercubes

In this subsection an algorithm is proposed, which extends the initial labeling of the leader's links to an orientation. The initial labeling and the availability of a leader uniquely define an orientation as expressed in the following theorem (given here without proof).

Theorem 4.1 *Let $\mathcal{L}$ be a labeling of the hypercube and w be a designated node. There exists exactly one orientation $\mathcal{O}$ which satisfies $\mathcal{O}_w(v) = \mathcal{L}_w(v)$ for each neighbor of w.*

The algorithm computes exactly this orientation, and, moreover, a corresponding labeling of the nodes with bitstrings of length n, where the leader is labeled with $(0, ..., 0)$.

```
begin rcvd_v := 0 ; dist_v := 0 ; label_v := (0, ..., 0) ;
      for l = 0 to n − 1 do (* Send for phase 1 *)
          begin send ⟨dim, l⟩ via link l ;
                π_v[l] := l
          end ;
      while rcvd_v < n do (* Receive for phase 2 *)
            begin receive ⟨label, l⟩ (* necessarily via link l *)
                  rcvd_v := rcvd_v + 1
            end
end .
```

Algorithm 9: ORIENTATION OF LEADER HYPERCUBE (LEADER).

This node labeling is also uniquely defined. The algorithm uses three types of messages. The leader sends to each of its neighbors the label of the connecting link in $\langle$**dim**, $i\rangle$ messages. Non–leaders send their node label to other processors in $\langle$ **iam**, $(b_0, ..., b_{n-1})\rangle$ message. Non–leaders inform their neighbors about the label of connecting links in $\langle$**label**, $i\rangle$ messages.

The algorithm is given as Algorithm 9 (for the leader) and Algorithm 10 (for non–leaders). It consists of two phases, where in the first phase messages flow away from the leader, and in the second phase messages flow towards the leader. In the sequel, let w denote the leader processor. A *predecessor* of node v is a neighbor u of v for which $d(u, w) < d(v, w)$, and a *successor* of node v is a neighbor u of v for which $d(u, w) > d(v, w)$. In a hypercube node v has no neighbor u for which $d(u, w) = d(v, w)$ and a node at distance d from the leader has d predecessors and $n - d$ successors.

The leader initiates the algorithm by sending a $\langle$**dim**, $i\rangle$ message over the link labeled i. When a non–leader processor v has learned its distance $dist_v$ to the leader and has received $dist_v$ messages from its predecessors, v is able to compute its node label. Processor v forwards this label in an $\langle$ **iam**, $(b_0, ..., b_{n-1})\rangle$ message to its successors. To show that v is indeed able to do so, first consider the case where v receives a $\langle$**dim**, $i\rangle$ message via link l. As the message is sent by the leader, $dist_v = 1$, and all other neighbors are successors. The node label of v is $(b_0, ..., b_{n-1})$, where $b_i = 1$, and the other bits are 0. (The label of link l becomes i in this case.) Thus v forwards $\langle$ **iam**, $(b_0, ..., b_{n-1})\rangle$ via all links $k \neq l$.

Next, consider the case where v receives an $\langle$ **iam**, $(b_0, ..., b_{n-1})\rangle$ message. The distance d of the sender of this message to the leader is derived from the message (the number of 1's in $(b_0, ..., b_{n-1})$). $\langle$ **iam**, *label* $\rangle$ messages are sent only to successors, thus the sender is a predecessor of v and $dist_v = d + 1$. As v has $dist_v$ neighbors at distance d, v waits until $dist_v$ $\langle$ **iam**, *label* $\rangle$ messages have been received. By then v computes its node label as the logical disjunction of the received node labels, and forwards it to the neighbors from which no $\langle$ **iam**, *label* $\rangle$ was received, as these are the successors.

In the first phase, each non–leader processor v computes its node label. In the second phase, each non–leader processor v learns from its successors the orientation of the links to the successors, and computes the orientation of the links to the predecessors. This information is sent over the link in $\langle$**label**, $i\rangle$ messages. A processor sends $\langle$**label**, $i\rangle$ messages to its predecessors as soon as it has received these messages from all successors,

```
begin rcvd_v := 0 ; dist_v := n + 1 ; label_v := (0, ..., 0) ;
      forall l do neigh_v[l] := nil ;
      while rcvd_v < dist_v do (* Receive for phase 1 *)
         begin receive msg via link l ; rcvd_v := rcvd_v + 1 ;
               (* msg is a ⟨dim, i⟩ or ⟨iam, (b_0, ..., b_{n-1})⟩ message *)
               if msg is ⟨dim, i⟩ then
                  begin dist_v := 1 ;
                        neigh_v[l] := (0, ..., 0) ; label_v[i] := 1
                        (* So now label_v = (0, .., 1, .., 0), with one 1 *)
                  end
               else
                  begin dist_v := 1 + # of 1's in (b_0, ..., b_{n-1}) ;
                        label_v := (label_v or (b_0, ..., b_{n-1})) ;
                        neigh_v[l] := (b_0, ..., b_{n-1})
                  end
         end ;
      (* Send for phase 1 *)
      forall l with neigh_v[l] = nil do
            send ⟨iam, label_v⟩ via link l ;
      while rcvd_v < n do (* Receive for phase 2 *)
         begin receive ⟨label, i⟩ via link l ;
               rcvd_v := rcvd_v + 1 ; π_v[l] := i
         end ;
      (* Send for phase 2 *)
      forall l with neigh_v[l] ≠ nil do
         begin π_v[l] := bit in which label_v and neigh_v[l] differ ;
               send ⟨label, π_v[l]⟩ via link l
         end
end .
```

Algorithm 10: ORIENTATION OF LEADER HYPERCUBE (NON–LEADER).

and then terminates. The leader terminates when ⟨**label**, i⟩ messages have been received from all neighbors.

The variables for processor v are: $rcvd_v$, the number of messages already received; $dist_v$, the distance to the leader (computed when the first message arrives, initialized to $n+1$); $label_v$, the node label computed by v; $neigh_v[0..n-1]$, an array holding the node labels of the predecessors of v; and π_v, to store the orientation.

Lemma 4.2 *The algorithm terminates in every processor.*

Proof. Using induction on d it is easily verified that all processors at distance at most d eventually send the messages for phase 1. For $d = 0$, only the leader itself has distance d to the leader and it may send the messages without receiving other messages first. Assume all processors at distance d to the leader send all messages for phase 1, and consider a processor v at distance $d+1$ from the leader. As all predecessors of v eventually send the phase 1 messages to v, v eventually receives one of these messages, and sets $dist_v := d+1$. When v has received the phase 1 messages from all of its $d+1$ predecessors, v sends phase 1 messages itself (to its successors).

Similarly it is shown that all processors send the messages of phase 2 and terminate. □

Lemma 4.3 *After termination $\mathcal{O} = \pi(\mathcal{L})$ is an orientation. For neighbors v and u, $label_v$ and $label_u$ differ exactly in bit $\mathcal{O}_v(u)$ (which is equal to $\mathcal{O}_u(v)$).*

Proof. According to Theorem 4.1 there exists exactly one orientation $\mathcal{O}$ and one corresponding node labeling $\mathcal{N}$ such that $\mathcal{O}_w(v) = \mathcal{L}_w(v)$ and $\mathcal{N}(w) = (0, .., 0)$.

In phase 1 the processors compute the node labeling $\mathcal{N}$, as is seen by using induction on the distance to the leader. Node w sets $label_w$ to $(0, ..., 0)$, which is $\mathcal{N}(w)$. Neighbor v of w sets $label_v$ to $(b_0, ..., b_{n-1})$, where b_i is 1 if the link from w to v is labeled i in w, and 0 otherwise. Thus $label_v = \mathcal{N}(v)$.

Now assume all nodes u at distance d from w compute $label_u = \mathcal{N}(u)$ and consider node v at distance $d+1$ from w. $\mathcal{N}(v)$ is a string of $d+1$ 1's and $n-d-1$ 0's. Node v has $d+1$ predecessors, and $\mathcal{N}(u)$ is found for predecessor u by changing one 1 in $\mathcal{N}(v)$ into a 0. Thus the conjunction of the $d+1$ labels $\mathcal{N}(u)$ is indeed $\mathcal{N}(v)$.

After phase 1, for predecessor u of v, $neigh_v[l] = label_u$ with $l = \mathcal{L}_v(u)$. In phase 2, v computes $\pi_v[l]$ as the bit in which $label_v$ and $neigh_v[l]$ differ, so that $label_v$ and $label_u$ differ exactly in bit $\pi_v[l]$, which is $\mathcal{O}_v(u)$ as required. The same label is used by u for the link, after u receives v's $\langle$**label**, $\pi_v[l]\rangle$ message. □

The properties of Algorithm 9/10 are summarized in the following theorem.

Theorem 4.4 *There exists a deterministic algorithm to orient leader hypercubes, which exchanges the asymptotically optimal number of $2E$ messages.*

The bit complexity. As the $\langle$ **iam**, *label* $\rangle$ messages of the algorithm consist of a node label, they contain a string of n bits. It will now be shown that the algorithm can be implemented using only messages of $O(\log n)$ bits. The $\langle$**dim**, $i\rangle$ and $\langle$**label**, $i\rangle$ messages contain a number between 0 and $n-1$ and thus contain $O(\log n)$ bits.

The algorithm does not need all information contained in the $\langle$ **iam**, *label* $\rangle$ messages. It suffices to transmit the number of 1's, the smallest index at which there is a 1, and the sum modulo n of the indexes for which there is a 1. For a node label $label = (b_0, ..., b_{n-1})$ define the weight, low, and index sum as $weight(label) = \#\{i : b_i = 1\}$; $low(label) = \min\{i : b_i = 1\}$; $ixsum(label) = (\sum_{b_i=1} i) \bmod n$. Finally, the *summary* is the tuple $smmry(label) = (weight(label), low(label), ixsum(label))$. The summary of a node is the summary of its node label.

Lemma 4.5 *Let v be a node at distance $d+1 \geq 2$ from w.*
(1) $dist_v = d+1$ can be derived from one summary of a predecessor of v.
(2) The summary of v can be computed from the $d+1$ summaries of v's predecessors.
(3) The node label of v can be computed from the summary of v and the $d+1$ summaries v's predecessors.
(4) The node label of a predecessor u of v can be computed from the node label of v and the summary of u.

Proof. (1) The computation of $dist_v$ is trivial as $weight(\mathcal{N}(u))$ equals $d(u, w)$.

(2) Now let $d+1$ summaries of predecessors of v be given. d of the $d+1$ summaries have *low* equal to $low(\mathcal{N}(v))$, while one summary has a higher *low* (the predecessor whose label is found by flipping the *first* 1 in $\mathcal{N}(v)$). This gives $low(\mathcal{N}(v))$, but also identifies the index sum $ixsum_0$ of a node label which differs from $\mathcal{N}(v)$ in position *low*. Thus $ixsum(\mathcal{N}(v)) = (ixsum_0 + low(\mathcal{N}(v))) \bmod n$. This completes the computation of $smmry(\mathcal{N}(v))$.

(3) Let $\mathcal{N}(v) = (b_0, ..., b_{n-1})$. The $d+1$ indices i for which $b_i = 1$ are found as $ixsum(\mathcal{N}(v)) - ixsum(\mathcal{N}(u)) \bmod n$ for the $d+1$ choices of u as a predecessor of v.

(4) For a predecessor u of v, $\mathcal{N}(u)$ is found by flipping the bit indexed $(ixsum(\mathcal{N}(v)) - ixsum(\mathcal{N}(u))) \bmod n)$ from 1 to 0 in $\mathcal{N}(v)$. □

It follows from Lemma 4.5 that it suffices in the orientation algorithm to send the summary of node label instead of the full label, and hence the algorithm can be implemented with messages of $O(\log N)$ bits. As the messages are used to assign different labels to $\Omega(n)$ links, the information in the messages cannot be compressed below $O(\log n)$ bits.

4.2 Named Hypercubes

The algorithm to orient leader hypercubes can be preceded by an election algorithm (cf. the construction of Theorem 2.3) and then yields an orientation algorithm for named networks with properties summarized in the following theorem.

Theorem 4.6 *There exists a deterministic algorithm to orient named hypercubes, which exchanges the asymptotically optimal number of $O(E)$ messages.*

An alternative algorithm is obtained when extinction is applied directly to Algorithm 9/10. Denote, as in Subsection 2.2.1, by *LNA* Algorithm 9/10 and by *NNA* the algorithm obtained when extinction is applied. Cf. Theorem 2.6, algorithm *NNA* has a worst case message complexity which is bounded by $N \times 2E$, which is $n4^n$. In this section it will be shown that the worst case message complexity of algorithm *NNA* is actually bounded by $n3^n$. The proof is due to Anneke A. Schoone.

Define the *face* of the hypercube *spanned by* two nodes w and v, denoted $face(u,\ v)$, as the set of nodes "between" w and v.

$$face(w,v) = \{u\ :\ d(w,u) + d(u,v) = d(w,v)\}$$

The face spanned by two nodes at distance d forms a d–dimensional hypercube itself. The same face is spanned by each pair of opposite nodes in this sub–hypercube.

Lemma 4.7 *The n–dimensional hypercube has 3^n faces.*

Proof. For each w there are $\binom{n}{d}$ nodes v at distance d of w, and each of these v defines a different face containing w of dimension d. However, each face of dimension d is found for 2^d different choices of w (and a suitable v), so the number of faces of dimension d is $(2^n \times \binom{n}{d})/2^d = 2^{n-d} \times \binom{n}{d}$. Thus the total number of faces is $\sum_{d=0}^{n} 2^{n-d} \times \binom{n}{d} = (1+2)^n = 3^n$. □

For the analysis of algorithm *NNA* cost is charged to the number of times a processor exits the receive loop of phase 1. If processor v does so in the execution of *LNA* initiated by w, one unit of cost is charged to $face(w, v)$.

Lemma 4.8 *Each face gets charged at most one cost unit.*

Proof. If processor v exits the receive loop of phase 1 in the execution of *LNA* which is initiated by w, then processor w has the largest name of all processors in $face(w, v)$. This is because all processors in the face must forward the messages carrying w's name. As only one processor w in the face has the largest name (in the face), and the same face is not spanned by w and a node other than v, the lemma follows. □

Theorem 4.9 *Algorithm NNA sends at most $n3^n$ messages in the worst case.*

Proof. In algorithm *LNA* each processor sends n messages, all of them after the waiting in phase 1 has terminated. Thus the number of messages sent in algorithm *NNA* is bounded by n times the number of cost units charged. The proof is completed using Lemmas 4.7 and 4.8. □

4.3 Anonymous Hypercubes

The algorithm in Subsection 4.1 can be combined with the election algorithm for anonymous networks in Subsection 2.2.1, because the number of nodes is known. This proves the following result.

Theorem 4.10 *There exists a randomized algorithm to orient anonymous hypercubes, which terminates with probability 1.*

5 The Orientation of Tori

In this section the problem of finding orientations for a torus is studied. It will be shown that deterministic algorithms for the orientation of leader and named tori exist, and that anonymous tori can be oriented by a randomized algorithm if and only if the size of the torus is known to the processors. For $n = 4$, the $n \times n$ torus is isomorphic to the 4–dimensional hypercube. An algorithm to orient it is easily obtained by application of the algorithm to orient the hypercube, followed by a local relabeling based on a processor's node label. The case $n = 3$ is simply ignored in this paper. In this section $n \times n$ tori are considered for $n \geq 5$, for a reason which will become clear in the next paragraph.

The orientation problem will be solved in two stages. The first stage computes a *consistent prelabeling*, and the second stage computes the orientation. The first stage is necessary because in torus networks it is in general *not* possible to extend the labeling of a single processor to a global orientation. In an oriented torus the *up* and *down* neighbor of one processor are at distance 2 of each other, and there exists *one* path of length 2 between them, provided $n \geq 5$. The same holds for the *right* and *left* neighbor. Each processor has pairs of neighbors which are at distance 2 of each other, for which there exist *two* paths of length 2 between these nodes (see Figure 11). It follows that a labelling which

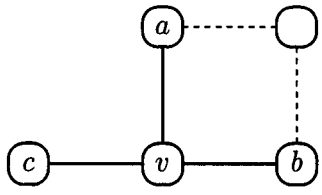

Figure 11: NEIGHBORS OF NODE v IN THE TORUS.

assigns the labels *up* and *down* to processors forming such a pair cannot be extended to an orientation.

The first stage of the algorithm divides the four links of each processor into two pairs, with the property that the links of one pair must have opposite labels in an orientation.

Definition 5.1 *A* prelabeling $\mathcal{P}$ *of the torus is an assignment in each node of labels from the set* $\{hori, verti\}$ *to the links of that node, such that each label is used twice.*
A prelabeling $\mathcal{P}$ *is* consistent *if for all nodes* v *and neighbors* u *and* w *of* v, $\mathcal{P}_v(u) = \mathcal{P}_v(w)$ *implies that there exists one path of length 2 between* u *and* w.

When a consistent prelabeling is available, a node may label the *verti* links with *up* and *down* and the *hori* links with *left* and *right*, and this labeling can be extended to an orientation. When a prelabeling is given, the *opposite* of a link is the single link with the same label, and the *perpendicular* links are the two links with different label.

5.1 Named Tori

On named tori both stages can be performed by deterministic algorithms.

```
begin forall links l do bag_v[l] := ∅ ;
      forall l do send ⟨ one, name_v ⟩ via l ;
      rcvd_v := 0 ;
      while rcvd_v < 16 do
         begin receive msg via link l ; rcvd_v := rcvd_v + 1 ;
               (* It is a ⟨ one, n ⟩ or ⟨ two, n ⟩ message *)
               if msg is a ⟨ one, n ⟩ message
                  then forall k ≠ l do send ⟨ two, n ⟩ via k
                  else bag_v[l] := bag_v[l] ∪ {n}
         end ;
      find l_1, l_2 such that bag_v[l_1] ∩ bag_v[l_2] = ∅ ;
      label l_1, l_2 with hori and the other links with verti
end .
```

Algorithm 12: PRELABELING A NAMED TORUS.

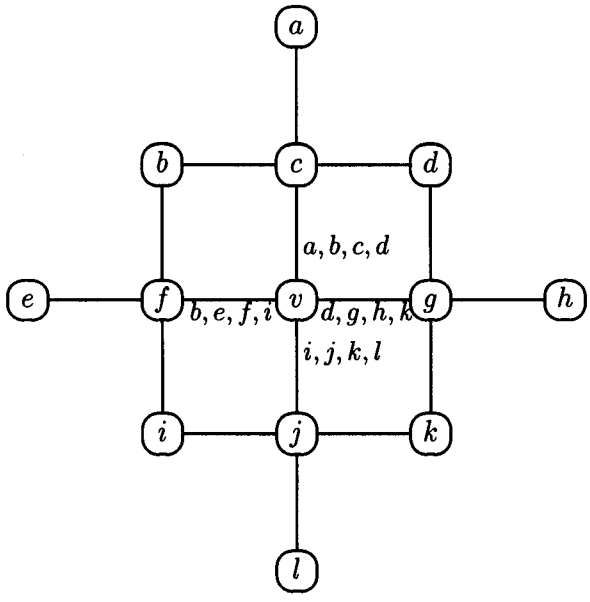

Figure 13: THE MESSAGES RECEIVED BY PROCESSOR v.

Computing a Prelabeling. The algorithm to compute a consistent prelabeling in a named network is given as Algorithm 12. Each processor sends its name to each neighbor and each neighbor forwards the name one step further. Thus, the name of each processor is transmitted through 16 links, and each processor receives 16 messages (4 through each link). Of the 16 messages received, 8 contain names which are received only once, and 4 contain names which are received twice, but through different links (see Figure 13).Two links via which the same name is received are perpendicular; so, when a processor has received its 16 messages a consistent prelabeling can be computed. The names are first sent in a $\langle$ **one**, n $\rangle$ message, and forwarded in a $\langle$ **two**, n $\rangle$ message. The names received in $\langle$ **two**, n $\rangle$ messages are stored in sets bag_v for each link.

Theorem 5.2 *There exists a deterministic algorithm to compute a consistent prelabeling on a named torus, which exchanges* $16N$ *messages.*

Computing an Orientation. An orientation can be computed by an algorithm based on the Spanning Tree algorithm by Gallager, Humblet, and Spira [GHS83]. A brief description of this algorithm follows. During the execution of the algorithm the network is partitioned into *fragments*, each with a *fragment name.* Initially, each fragment consists of a single node, and the name of the fragment is the name of the node.

During the execution fragments are enlarged because fragments combine and form new, larger fragments. To this end the processors in a fragment cooperate to select one link which leaves the fragment as the *preferred link* of the fragment and send a **connect** message via this link. Eventually the fragment at the other end of the preferred link agrees to connect the two fragments, after which a new, larger fragment is formed. The name

For processor v:
send $\langle$**answer**, $\pi_v[l]\rangle$ via l ; (* to u *)
send $\langle$ **dione**, $\pi_v[k]\rangle$ via one link k perpendicular to l

For processor u:
receive $\langle$**answer**, $ll\rangle$ via l ; (* from v *)
$\sigma[l] := oppos(ll)$; (* The opposite direction *)
$\sigma[oppos(l)] := ll$;
send $\langle$ **direq** $\rangle$ via the two links perpendicular to l ;
receive $\langle$ **dians**, $kk\rangle$ via link k ;
$\sigma[k] := kk$; $\sigma[oppos(k)] := oppos(kk)$
(* Now σ is the permutation to be applied to all
link labels in the fragment. *)

For all other processors:
when a $\langle$ **dione**, $ll\rangle$ message is received via link l:
send $\langle$ **ditwo**, $ll\rangle$ via the two links perpendicular to l

when a $\langle$ **ditwo**, $ll\rangle$ message is received via link l
and a $\langle$ **direq** $\rangle$ message via link k:
send $\langle$ **dians**, $ll\rangle$ via link k

Algorithm 14: FRAGMENT COMBINE PROTOCOL.

of the new fragment is chosen to be the name of one of the combining fragments[2]. This name is flooded towards all nodes in the old fragment of which the name is not chosen to be the new name.

The algorithm terminates when the entire network consists of a single fragment. The name of the fragment to which processor v belongs is $frname_v$.

The orientation algorithm is found as an extension of the Spanning Tree algorithm. It is ensured, that processors which share the same fragment name, also share the same orientation. To this end, when the new fragment name is flooded to the processors that must update fr_v, also the relative orientation of the two fragments is flooded. It must be shown how this relative orientation is computed.

The combining of fragments is embedded in the protocol given in Algorithm 14. Assume node v in fragment F answers the **connect** message of node u in fragment G with an **answer** message, where the combined fragment will have the name (and orientation) of fragment F. Processor v includes in the answer the label of the link over which it is sent, and this defines for u the new label of the link over which the message is received, as well as the opposite link. It remains to find the correct orientation of the two perpendicular links. To do this (see Figure 15), v sends over one link l perpendicular to link (v, u) a message $\langle$ **dione**, $\pi_v[l]\rangle$, which is forwarded by the receiving node as a $\langle$ **ditwo**, $\pi_v[l]\rangle$ message. Processor u sends a $\langle$ **direq** $\rangle$ message via the two links perpendicular to link (u, v). A processor which receives both a $\langle$ **ditwo**, $ll\rangle$ and a $\langle$ **direq** $\rangle$ message, replies to

[2]In [GHS83] it occurs that a new name is chosen for a new fragment, but the algorithm can easily be modified so as to use an existing name for the new fragment.

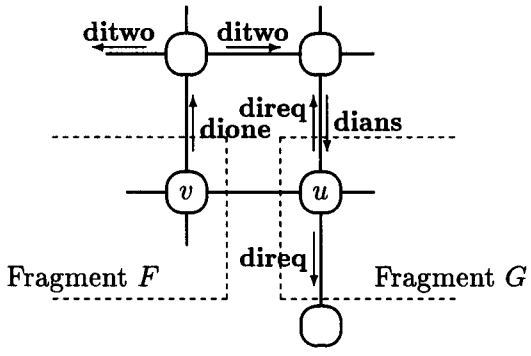

Figure 15: EXTRA MESSAGES IN THE CONNECTING PROCEDURE.

the ⟨ **direq** ⟩ message with a ⟨ **dians**, ll ⟩ message. Thus u receives an answer to one of its ⟨ **direq** ⟩ messages, which gives the orientation of the links perpendicular to (u, v).

To distinguish between the messages of different invocations of the connect protocol, all messages are tagged with the fragment names of F and G (not shown in Figure 15). The connect protocol to combine two fragments exchanges 6 messages each time two fragments are combined, and as exactly $N - 1$ merges take place, the message complexity of the orientation protocol exceeds the complexity of the underlying algorithm by $6N - 6$ messages.

Thus the complexity of the second stage is $O(N \log N)$, which exceeds the complexity of the first stage in order of magnitude.

Theorem 5.3 *There exists a deterministic algorithm for the orientation of named tori, which exchanges $O(N \log N)$ messages in the worst case.*

5.2 Leader Tori

Computing a Prelabeling. A preorientation can be deterministically computed only by a computation starting from the leader. This computation could start by applying the naming algorithm (Algorithm 3/4), followed by Algorithm 12.

Theorem 5.4 *There exists a deterministic algorithm to compute a consistent prelabeling on a leader torus, which exchanges $2E + N - 1 + 16N = 21N - 1$ messages.*

Computating an Orientation. As the computation of a consistent prelabeling includes the assignment of names, stage 2 can be performed as for named tori, which would cost $O(N \log N)$ messages. Using the same ideas as for the connect protocol, Algorithm 14, it is possible to give an algorithm which exchanges only $O(N)$ messages. The details of this algorithm are left as an exercise for the reader.

Theorem 5.5 *There exists a deterministic algorithm for the orientation of leader tori, which exchanges $O(N)$ messages.*

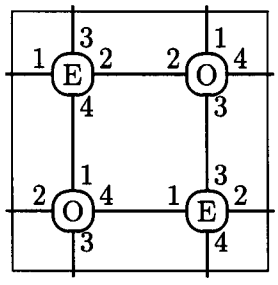

Figure 16: A SYMMETRIC LABELING WHICH IS NOT A PRE–PREORIENTATION.

5.3 Anonymous Tori

Although in an anonymous network a consistent prelabeling can be computed (by a randomized algorithm), it is impossible to compute an orientation, even by use of a randomized algorithm, when N is not known.

Computing a Prelabeling. A consistent prelabeling can be computed in an anonymous torus, but only by a randomized algorithm if the size of the torus is even.

Theorem 5.6 *There exists no deterministic algorithm to compute a consistent prelabeling for an anonymous torus of even size.*

Proof. The proof uses the same techniques as the proofs in Subsection 2.4. A labeling is a pre–prelabeling if it can be turned into a consistent prelabeling by the application of a fixed function ρ_0 from $\{up, down, left, right\}$ to $\{hori, verti\}$. A symmetric labeling which is not a pre–prelabeling is found by covering the torus with label patterns as in Figure 16. In this labeling, the even nodes (E) must label links 1 and 2 the same, while the odd nodes (O) must assign the same label to links 2 and 4. □

A consistent prelabeling can be computed by a randomized algorithm which is an extension of Algorithm 12. Processors cannot send their name, but instead draw a random number (in the range $[1, ..., M]$, say) and send this number together with a phase number (initially 1). Processors receiving 12 $\langle$ **two**, $n, p \rangle$ messages, but not carrying four numbers twice and four numbers once, reply by sending $\langle$ **refuse**, $n, p \rangle$ messages. These messages are sent back to the processors from which the number n originated and causes them to draw a new number in the next phase.

The probability that a "collision" occurs in a processor can be made small by choosing M large. The precise formulation of the algorithm, as well as the analysis of its expected message complexity is left to the reader.

Theorem 5.7 *There exists a randomized algorithm to compute a consistent prelabeling for an anonymous torus.*

Computing an Orientation. It has been established in Subsection 2.4 that anonymous tori cannot be oriented by deterministic algorithms. The results presented so far suffice to

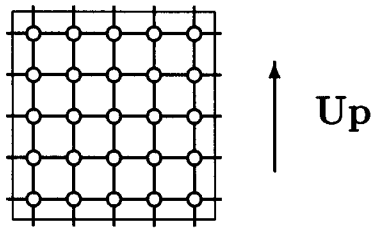

Figure 17: ALGORITHM A TERMINATES CORRECTLY ON A SMALL TORUS.

construct a randomized algorithm for the orientation of tori when the size of the network is known. To this end, Algorithms 5, 3/4, and the algorithm sketched in Subsection 5.1 can be combined.

Theorem 5.8 *There exists a randomized algorithm for the orientation of anonymous tori, when the number of processors is known.*

The main result of this subsection is to prove that no such algorithm exists when the size of the torus is not known. The proof relies on techniques similar to those used by Itai and Rodeh [IR81] to establish that no (randomized) algorithm exists to compute the size of a anonymous ring network. An execution leading to a correct result on a (small) torus is finite, and hence it has a positive probability of being "accidently simulated" by a fragment of a larger torus. If this occurs in two different parts of the larger torus, processes may terminate with incompatible orientations, and this may happen with an arbitrarily large probability.

Theorem 5.9 *There exists no (randomized) algorithm for the orientation of tori when the number of processors is not known.*

Proof. Assume there exists an algorithm A that is able to compute an orientation in an $n_0 \times n_0$ torus T_0 (see Figure 17). That is, there exists an execution Ex of A on the $n_0 \times n_0$ torus in which every processor terminates, and the resulting labeling is an orientation. Define a message chain as a series of messages $(M_1, M_2, ..., M_k)$, such that M_{i+1} was sent by the processor that received M_i, and was sent only after the receipt of M_i. Let the longest message chain in Ex have length L.

Next consider a torus T_1 of size $n_1 \times n_1$ with $n_1 > 2L + 1$. The *l–neighborhood* of processor v is the set of processors with distance at most l to v. Processor (i, j) in T_1 *corresponds* to processor $(i \bmod n_0, j \bmod n_0)$ in T_0. In execution Ex, all processors of T_0 take finitely many steps, and in particular draw a random number only finitely often. Thus, there is an $\epsilon > 0$ such that for each processor v_0 of T_1 the probability that all processors in the L–neighborhood of v_0 draw the same numbers as the corresponding processor in T_0 draw in Ex is at least ϵ. If this happens there is an execution of A on T_1 in which processor v_0 terminates after executing exactly the same steps as in execution Ex (see Figure 18).

The size of T_1 can be chosen large enough to have an arbitrarily high probability that this occurs for at least one processor v_0. The size of T_1 can be chosen large enough to

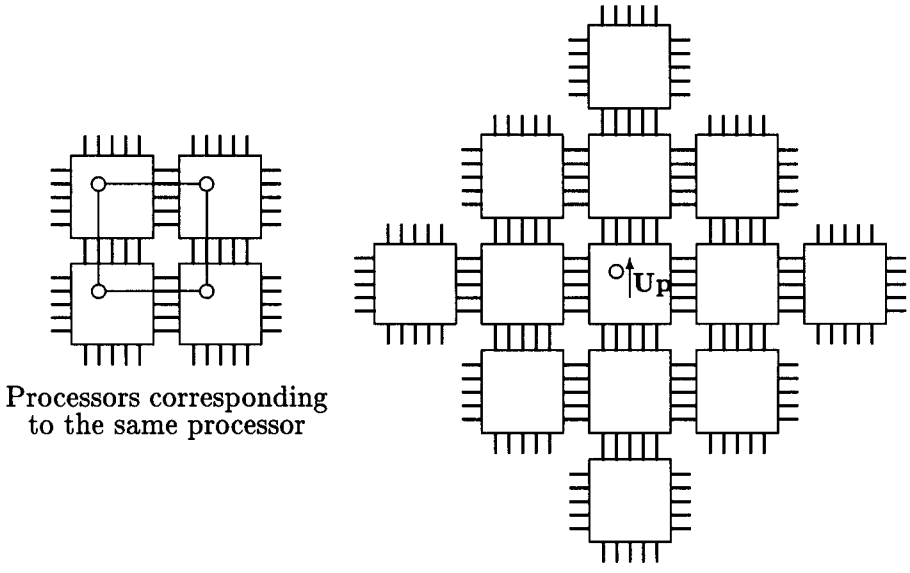

Figure 18: A SUBSET OF THE PROCESSORS SIMULATES *Ex*.

have an arbitrarily high probability of this to happen for at least 2 processors v_0 and v_1, where v_0 and v_1 terminate with different orientations (see Figure 19). □

6 Alternative Characterization of Orientations

The definitions of orientations given in Section 2.1 were based on the existence of particular (global) name assignments to nodes. By (implicitly) computing such a global name assignment, all orientation algorithms in this paper implicitly elect a leader.

Breaking symmetry, however, is a non–trivial task even in oriented networks (even though in oriented networks it is easier than in unoriented networks). It is thus to be

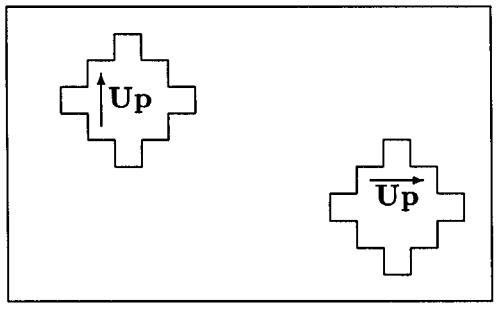

Figure 19: PROCESSORS TERMINATE WITH INCONSISTENT LABELINGS.

expected that breaking symmetry is a "harder" task than finding an orientation, and that more efficient or simpler algorithms can be found which compute an orientation without breaking the symmetry. To prove the correctness of such algorithms it would be necessary to express the correctness of an orientation without making reference to a corresponding node labeling. A characterization of orientations, based only on the link labels, is derived in this section.

Orientations can be characterized by the properties of the label strings of closed paths, as defined below. A *path* (of *length* k) in a network is a sequence of nodes $v_0, v_1, .., v_k$, such that for each i, v_i is connected to v_{i+1}. A path $v_0, v_1, .., v_k$ is *closed* if $v_0 = v_k$. For a path $P = v_0, v_1, .., v_k$ in a labeled network, the *label string* $LS(P)$ is the sequence $\mathcal{L}_{v_0}(v_1), \mathcal{L}_{v_1}(v_2), ..., \mathcal{L}_{v_{k-1}}(v_k)$. The *reverse* of a path $P = v_0, v_1, .., v_k$ is the path $P^R = v_k, ..., v_1, v_0$. The concatenation of paths $P_1 = v_0, v_1, .., v_k$ and $P_2 = w_0, w_1, ..., w_l$, where $v_k = w_0$, is the path $P_1P_2 = v_0, v_1, .., v_k, w_1, ..., w_l$.

6.1 The Clique

Let a labeling $\mathcal{L}$ in a clique be given. For a path $P = v_0, v_1, .., v_k$, let $Sum(P) = \sum_{i=0}^{k-1} \mathcal{L}_{u_i}(u_{i+1})$, that is, the sum of the labels in the label string of P, where addition is modulo N.

Theorem 6.1 *The following two are equivalent:*
(1) $\mathcal{L}$ is an orientation.
(2) A path P is closed if and only if $Sum(P) = 0$.

Proof. Assume 1. By the definition of an orientation there exists a labeling $\mathcal{N}$ of the nodes such that $\mathcal{L}_u(v) = \mathcal{N}(v) - \mathcal{N}(u)$. Using induction on the path, it is easily shown that $Sum(P) = \mathcal{N}(v_k) - \mathcal{N}(v_0)$. As labels are from the set $\{1, .., N-1\}$, $\mathcal{L}_u(v) \neq 0$, and all node labels are different. It now follows that $v_0 = v_k$ if and only if $Sum(P) = 0$.

Assume 2. Pick an arbitrary node v_0, set $\mathcal{N}(v_0) = 0$ and for all $u \neq v_0$ set $\mathcal{N}(u) = \mathcal{L}_{v_0}(u)$. It remains to show that this node labeling satisfies the constraints in the definition of an orientation. That is, for all nodes u_1 and u_2, $\mathcal{L}_{u_1}(u_2) = \mathcal{N}(u_2) - \mathcal{N}(u_1)$. For $u \neq v_0$, note $\mathcal{N}(v_0) = 0$ and $\mathcal{N}(u) = \mathcal{L}_{v_0}(u)$ so $\mathcal{L}_{v_0}(u) = \mathcal{N}(u) - \mathcal{N}(v_0)$. As $P = v_0, u, v_0$ is closed, $Sum(P) = 0$, hence $\mathcal{L}_u(v_0) = -\mathcal{L}_{v_0}(u) = \mathcal{N}(v_0) - \mathcal{N}(u)$. For $u_1, u_2 \neq v_0$, as $P = v_0, u_1, u_2, v_0$ is closed, $Sum(P) = 0$ and hence $\mathcal{L}_{u_1}(u_2) = -\mathcal{L}_{v_0}(u_1) - \mathcal{L}_{u_2}(v_0) = (\mathcal{N}(u_1) - \mathcal{N}(v_0)) - (\mathcal{N}(v_0) - \mathcal{N}(u_2)) = \mathcal{N}(u_2) - \mathcal{N}(u_1)$. □

6.2 The Hypercube

Let a labeling $\mathcal{L}$ in a hypercube be given. For a path $P = v_0, v_1, .., v_k$, let $\#_i(P)$ be the number of i's in $LS(P)$.

Theorem 6.2 *The following two are equivalent:*
(1) $\mathcal{L}$ is an orientation.
(2) A path P is closed if and only if for all i : $\#_i(P) = 0 \pmod 2$.

Proof. Assume 1. By definition, each node v can be assigned a unique name $\mathcal{N}(v) = (b_0, b_1, .., b_{n-1})$, such that the edge (v, w) is labeled i in v if $\mathcal{N}(w) = (b_0, .., \bar{b_i}, .., b_{n-1})$. It follows by induction on the path, that with $\mathcal{N}(v_0) = (b_0, b_1, .., b_{n-1})$ $\mathcal{N}(v_k) = (c_0, c_1, .., c_{n-1})$,

where $c_i = (b_i + \#_i(P)) \bmod 2$. Hence $v_0 = v_k$ if and only if $\forall i : c_i = b_i$, i.e., $\forall i : \#_i(P) \bmod 2 = 0$.

Assume 2. First note that for each edge (v, w), $P = v, w, v$ is a closed path, which implies (by 2) that $\mathcal{L}_v(w) = \mathcal{L}_w(v)$. This implies that $\#_i(P) = \#_i(P^R)$.

Next a node naming function $\mathcal{N}$ is defined. Pick an arbitrary node v_0. For a node u, let P be any path from v_0 to u and set $\mathcal{N}(u) = (c_0, c_1, .., c_{n-1})$, where $c_i = \#_i(P) \bmod 2$.

It must now be shown that this definition is sound. Let P_1 and P_2 be two paths from v_0 to u. Now $P_1 P_2^R$ is a closed path, so $\#_i(P_1 P_2^R) = 0 \pmod 2$. Because $\#_i(P_1 P_2^R) = \#_i(P_1) + \#_i(P_2)$ this implies that $\#_i(P_1) \bmod 2 = \#_i(P_2) \bmod 2$, so $\mathcal{N}(u)$ is independent of the choice of a path from v_0 to u and the name function is well defined.

Next it is shown that names are unique. Assuming $\mathcal{N}(u_1) = \mathcal{N}(u_2)$, let P_1 and P_2 be paths from v_0 to u_1 and u_2. Consider the path $P_1^R P_2$ from u_1 to u_2. As $\#_i(P_1) \bmod 2 = \#_i(P_2) \bmod 2$, $\#_i(P_1^R P_2) = 0 \pmod 2$ for all i. It follows by (2) that $P_1^R P_2$ is closed, so $u_1 = u_2$.

Finally it must be shown that for a node v with $\mathcal{N}(v) = (b_0, b_1, .., b_{n-1})$, $\mathcal{L}_v(w) = i$ for the neighbor w with $\mathcal{N}(w) = (b_0, .., \bar{b_i}, .., b_{n-1})$. Let P be a path from v_0 to v, then $P' = P, w$ is a path from v_0 to w. As the label string of P' is just the label string of P extended with $\mathcal{L}_v(w)$ (that is, i), it follows that $\mathcal{N}(w) = (b_0, .., \bar{b_i}, .., b_{n-1})$. □

6.3 The Torus

Let a labeling $\mathcal{L}$ in a torus be given. For a path $P = v_0, v_1, .., v_k$, let $\#_{up}(P)$ be the number of *up*'s in $LS(P)$, and let $\#_{down}$, etc., be defined similarly.

Theorem 6.3 *The following two are equivalent:*
(1) $\mathcal{L}$ is an orientation.
(1) A path P is closed if and only if $\#_{up}(P) - \#_{down}(P) = 0 \pmod n$ and $\#_{right}(P) - \#_{left}(P) = 0 \pmod n$.

Proof. Assume 1. By definition, each node v can be assigned a unique name $\mathcal{N}(v) = (i, j)$, such that the edge (v, w) is labeled *up* (*down*, *right*, *left*) in v if $\mathcal{N}(w) = (i, j+1)$ ($(i, j-1)$, $(i+1, j)$, $(i-1, j)$). It follows by induction on the path, that, with $\mathcal{N}(v_0) = (i, j)$, $\mathcal{N}(v_k) = (i + \#_{up}(P) - \#_{down}(P) \bmod n, j + \#_{right}(P) - \#_{left}(P) \bmod n)$. Hence $v_0 = v_k$ if and only if $\#_{up}(P) - \#_{down}(P) = 0 \pmod n$ and $\#_{right}(P) - \#_{left}(P) = 0 \pmod n$.

Assume 2. First note that for each edge (v, w), $P = v, w, v$ is a closed path, hence (by 2) $\#_{up}(P) - \#_{down}(P) = 0 \pmod n$ and $\#_{right}(P) - \#_{left}(P) = 0 \pmod n$. It follows that if $\mathcal{L}_v(w) = up$ (*down*, *right*, *left*) then $\mathcal{L}_w(v) = down$ (*up*, *left*, *right*). This implies that $\#_{up}(P) = \#_{down}(P^R)$, and so on.

Next a node naming function $\mathcal{N}$ is defined. Pick an arbitrary node v_0. For a node u, let P be a path from v_0 to u and set $\mathcal{N}(u) = (\#_{up}(P) - \#_{down}(P) \bmod n, \#_{right}(P) - \#_{left}(P) \bmod n)$.

It must now be shown that this definition is sound. Let P_1 and P_2 be two paths from v_0 to u. Now $P_1 P_2^R$ is a closed path, so $\#_{up}(P_1 P_2^R) - \#_{down}(P_1 P_2^R) = 0 \pmod n$ and $\#_{right}(P_1 P_2^R) - \#_{left}(P_1 P_2^R) = 0 \pmod n$. Because $\#_{up}(P_1 P_2^R) = \#_{up}(P_1) + \#_{down}(P_2)$ and $\#_{down}(P_1 P_2^R) = \#_{down}(P_1) + \#_{up}(P_2)$, the former implies $\#_{up}(P_1) - \#_{down}(P_1) \bmod n = \#_{up}(P_2) - \#_{down}(P_2) \bmod n$. Similarly, the latter implies $\#_{right}(P_1) - \#_{left}(P_1) \bmod$

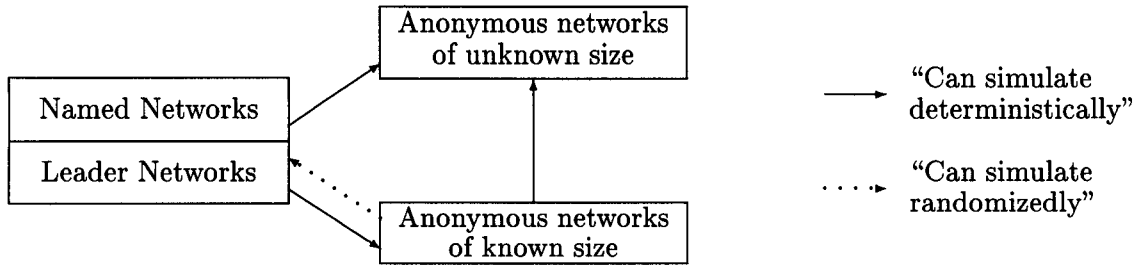

Figure 20: LEADER, NAMED AND ANONYMOUS NETWORKS CAN SIMULATE EACH OTHER.

$n = \#_{right}(P_2) - \#_{left}(P_2) \bmod n$. It follows indeed, that $\mathcal{N}(u)$ is independent of the choice of a path from v_0 to u and the name function is well defined.

Next it is shown that names are unique. Assuming $\mathcal{N}(u_1) = \mathcal{N}(u_2)$, let P_1 and P_2 be paths from v_0 to u_1 and u_2. Consider the path $P_1^R P_2$ from u_1 to u_2. As $\#_{up}(P_1) - \#_{down}(P_1) \bmod n = \#_{up}(P_2) - \#_{down}(P_2) \bmod n$, it follows that $\#_{up}(P_1^R P_2) - \#_{down}(P_1^R P_2) = 0 \pmod n$. Similarly, $\#_{right}(P_1^R P_2) - \#_{left}(P_1^R P_2) = 0 \pmod n$. It follows by (2) that $P_1^R P_2$ is closed, so $u_1 = u_2$.

Finally it must be shown that for a node v with $\mathcal{N}(v) = (i,j)$, $\mathcal{L}_v(w) = up$ for the neighbor w with $\mathcal{N}(w) = (i, j+1)$. Let P be a path from v_0 to v, then $P' = P, w$ is a path from v_0 to w. As the label string of P' is just the label string of P extended with $\mathcal{L}_v(w)$ (that is, *up*), it follows that $\mathcal{N}(w) = (i, j+1)$. The case for *down*, *right*, and *left* is handled similarly. □

7 Discussion

In this paper the problem of finding orientations for two network topologies has been studied under three model assumptions. The results of the study can be summarized as follows. The problem of network orientation can be solved by a deterministic algorithm in leader or named networks. The problem cannot be solved by a deterministic algorithm in anonymous networks. In anonymous networks the problem can be solved by a randomized algorithm if the size of the network is known (which is the case for cliques and hypercubes), and cannot be solved by a randomized algorithm when the size is not known.

These results are in accordance with know results in the area of distributed computing, cf. Figure 20. Named networks can simulate leader networks (Theorem 2.3) and vice versa (Theorem 2.2). Anonymous networks can simulate leader networks with a randomized algorithm when the network size is known (Theorem 2.11), but not when the size is unknown.

7.1 Dependency of other assumptions

In this paper the solvability of the orientation problem was studied as a function of the required *symmetry* of the solution. The solvability may as well be studied as a function of other parameters.

7.1.1 Fault–Tolerance

In this paper is was assumed that the network were *reliable*, that is, processors and links do not fail. Algorithms research in the past decade has frequently addressed the question if processors can be coordinated in systems where processors can fail, for example, according to one of the following fault models.

- **Initially Dead Processors:** It may occur that some processors do not execute a single instruction of their local algorithm.
- **Crashes:** It may occur that some processors stop executing their local algorithm at arbitrary moments in the execution.
- **Byzantine Faults:** It may occur that some processors execute steps which are in disaccordance with their local algorithm, such as sending messages with wrong information.

A result of Moran and Wolfstahl [MW87] indicates that no deterministic orientation algorithm exists that is resilient to a crash of a single processor. This leaves open, whether randomized solutions could tolerate processor crashes or even Byzantine faults. The results of Fischer, Lynch, and Peterson [FLP85] indicate, that deterministic algorithms can coordinate non–trivial decisions in the presence of initially dead processors. This suggests the question whether deterministic algorithms exist for the orientation of networks in the presence of initially dead processors.

7.1.2 Refined Symmetry Assumptions

In this paper only three different symmetry assumptions were considered, namely that all local algorithms are different (named networks), all local algorithms are identical (anonymous networks), or all local algorithms except one are identical (leader networks). Different assumptions about the symmetry are possible.

- k**–Leader:** There are exactly k processors that execute the leader algorithm, and all others execute the non–leader algorithm.
- **Difference:** There are two (or: k) different local algorithms, and each of them is executed by at least one processor.
- **[Maximally] Independent Leaders:** The set of processors executing the leader algorithm constitute a [Maximal] Independent set.

For each of these symmetry assumptions it can be investigated whether the class of computable functions is [strictly] included in or [strictly] includes the functions computable by leader or anonymous networks.

Open Question 7.1 *Fit the computational power of these symmetry assumptions in Figure 20.*

7.1.3 Synchronism Assumptions

In this paper *asynchronous systems* were considered. In these systems there is no bound on the time necessary to perform one operation, and no bound on the time between sending an receiving a message. A different model, which has frequently been used for the development of distributed algorithms, is that of *synchronous systems.* In synchronous systems bounds are known both on the time to perform one instruction and on the message delay time. The following four models can be distinguished.

- **Fully Asynchronous Networks:** The model that is considered in this paper.
- **Archimedean Networks:** Bounds on the relative speeds of components do exist; they can be very rough, however, and need not be known to the processors; see Vitanyi [Vit85].
- **Asynchronous Bounded Delay Networks:** Processing time within a processor is assumed to be neglictible, an upper bound is known on the message delay, and processors have clocks that run at the same speed (barring a very small drift); see, *e.g.*, Korach *et al.* [KTZ88].
- **Fully Synchronous Networks:** Processors execute their local algorithm in discrete rounds, and a message sent in round i is received *before* the receiver executes round $i+1$.

A lot of research has addressed the influence of synchronism assumptions on the functions that are computable, and the efficiency with which they can be computed.

The Power of Synchronism. *Stronger synchronism assumptions do* not *increase the class of functions computable by reliable networks.* This statement follows from the existence of so–called "synchronizer" algorithms, implementing fully synchronous networks on networks satisfying a weaker assumption. Awerbuch [Awe85] proposed a synchronizer for fully asynchronous networks, and his "α–synchronizer" can be used even for anonymous (but reliable) networks. A more efficient algorithm (in terms of messages exchanged by the synchronizer) was presented by Korach *et al.* [KTZ88] for Asynchronous Bounded Delay Networks.

The Efficiency of Synchronism. *Stronger synchronism assumptions allow more efficient algorithms.* This statement can be illustrated by some results on the complexity of electing a leader on a named ring network. It was shown (by various authors, *e.g.*, Pachl *et al.* [PKR84]) that on an asynchronous ring at least $\Omega(N \log N)$ messages must be exchanged. Vitanyi [Vit85] has demonstrated that on an Archimedean ring $O(N)$ messages suffice to elect a leader. The implicit constant hidden in the big–Oh notation depends on the ratio between the various upper and lower bounds on the relative speeds of components. Bodlaender and Tel [BT90] have shown, that on a synchronous ring $O(N)$ messages suffice, each message can be of $O(1)$ bits, and this is regardless of whether the processors know N or not.

A striking example of the efficiency that can be obtained from synchronism is the surprizing result that in a synchronous system any message M can be transmitted using $O(1)$ bits. This can be done by "coding M in time", namely, sending a **start** message

and M time units later a **stop** message. The receiver obtains M by measuring the time between the receipt of the two messages; see, *e.g.*, [BT90].

Fault–Tolerance and Synchronism. *Stronger synchronism assumptions are able to tolerate larger classes of faults in unreliable networks.* In a landmark paper, Fischer *et al.* [FLP85] have shown that no non–trivial agreement can be deterministically reached between processors in an asynchronous network in the possible presence of a single crash fault. On the other hand, Lamport *et al.* [LSP82] have shown that in a synchronous system agreement can be reached even in the presence of (up to almost $N/3$) Byzantine faults. These results show that no deterministic synchronizer algorithm exists for fully asynchronous systems where processor crashes may occur. The implementation of fault–tolerant systems usually relies on the availability of clocks and an upper bound on message delays (the Asynchronous Bounded Delay assumption). The fault–tolerant synchronization of clocks (see Ramanathan *et al.* [RSB90] for an overview article) is an important step in the implementation of a fully synchronous network.

7.2 Other Topologies

In this paper the orientation problem was studied for cliques, hypercubes and tori. The problem can similarly be defined for other network topologies, such as shuffles, cube connected cycles, or multi dimensional grids. Orientations can easily be defined for these specific topologies, as was done for cliques, hypercubes, and tori.

Kranakis and Krizanc [KK90b] define *Cayley networks* as follows. Let $\mathcal{G}$ be a (finite) group generated by $\{g_1, \ldots, g_k\}$. The Cayley network of $\mathcal{G}$ is the graph $G = (V, E)$ where $V = \mathcal{G}$ and $E = \{(x, y) \mid \exists i \ : \ x = g_i y \vee y = g_i x\}$. The network topologies considered in this paper are special cases of Cayley networks, obtained by substituting for $\mathcal{G}$ groups with a relatively simple structure. More complicated groups give rise to different network topologies. Cayley networks can be naturally oriented by defining $\mathcal{O}_x(g_i x) = i$ and $\mathcal{O}_{g_i x}(x) = \overline{i}$. The related orientation problems may give rise to complicated algorithms, utilizing a large collection of algorithmical ingredients.

It is not clear whether the notion of orientations can be generalized to more general classes of topologies, for example, the class of all regular graphs. Orientations of planar graphs can be defined naturally. A labeling of a planar graph is an assignment in each node v of numbers from 1 to dgr_v to the edges of v. A labeling is an orientation if there exists a planar embedding of the graph, such that for each node the link labels increase in clockwise order.

Open Question 7.2 *Develop algorithms for the orientation of planar networks.*

7.3 Termination and Termination Detection

This subsection discusses two different notions of termination, namely *processor termination* and *message termination.* Message terminating algorithms are simpler to design and can compute a larger class of functions. A brief discussion of the *termination detection problem* is included.

```
var a_v, b_v : integer ; (* Input, result *)

begin b_v := a_v ;
      forall links k do send ⟨b_v⟩ via k ;
      while true do
        begin receive ⟨b⟩ ;
              if b > b_v then
                begin b_v := b ;
                      forall links k do send ⟨b_v⟩ via k
                end
        end
end .
```

Algorithm 21: COMPUTING THE MAXIMUM IN AN ANONYMOUS NETWORK.

Processor and Message Termination. The results in this paper are derived for *processor terminating* algorithms. In these algorithms eventually a system configuration is reached in which all processors are in a terminated state. (Such a configuration is reached in all executions of a deterministic algorithm, and with probability 1 in a randomized algorithm.) In such a state, a processor is unable to execute further steps of the algorithm, and the values of its variables in that state are the output of the problem.

An algorithm is *message terminating* if eventually a configuration is reached where no further step of the algorithm can be taken, *i.e.*, all processors are either in a terminated state, or waiting to receive but there are no messages in the channels. (Such a configuration is reached in all executions of a deterministic algorithm, and with probability 1 in a randomized algorithm.) In a waiting state a processor is able to receive a message of the algorithm, which would change the value of its variables. In a message terminated configuration such a message will of course never arrive, but message termination is a property of the global configuration and is unobservable to a single processor. Hence a processor is not aware that its variable have converged to their final values (the "output" of the algorithm).

It has turned out, that message terminating algorithms are often simpler to design and verify, because aspects related to process termination can be ignored.

The Power of Termination. Itai and Rodeh [IR81] have shown that in anonymous networks message terminating algorithms are able to compute a larger class of fuctions that processor terminating algorithms. An illustration of this result is found by considering the following problem. Each processor v in an anonymous network of unknown size has an input a_v, and it is required to compute in each processor the maximum over all inputs.

This computation can be carried out by a (deterministic) message terminating algorithm as stated in the following theorem.

Theorem 7.3 *Algorithm 21 terminates after exchanging at most NE messages. When the algorithm terminates, $b_v = \max_w a_w$ for each processor v.*

The proof is left as an exercise. On the other hand, the following impossibility result can be shown by methods similar to those used in the proof of Theorem 5.9.

Theorem 7.4 *There exists no (randomized) processor terminating algorithm to compute the maximum of the inputs in an anonymous network of unknown size.*

Corollary 7.5 *The class of functions computable by message terminating algorithms is* strictly *larger than the class of functions computable by processor terminating algorithms, for anonymous networks of unknown size.*

Proof. That the first class includes the second class follows because a processor terminating algorithm is also message terminating. The strictness follows from the previous two theorems. □

Termination Detection. Just like algorithms for leader networks can be used for named networks by the application of an election algorithm, message terminating algorithms can be made processor terminating by the application of a *termination detection* algorithm. A termination detection algorithm runs concurrently with an arbitrary message terminating algorithm. When the latter algorithm reaches a message terminated configuration, the former algorithm eventually detects this and sends a terminate message to all processors.

The design of termination detection algorithms has received a lot of attention during the past decade. There do exist termination detection algorithms for leader networks (Dijkstra and Scholten [DS80]) and named networks (Tan and Van Leeuwen [TL86]). The existence of a termination detection algorithm for anonymous networks of known size follows from the results in this paper or [Tel94, Sec. 8.3.4]. Corollary 7.5 implies that no termination detection algorithm exists for anonymous networks of unknown size.

Acknowledgements. The members of the Utrecht Distributed Algorithms Group are acknowledged for their stimulating discussions of the subject. I want to thank Anneke Schoone for her proof of Theorem 4.9 and Petra van Haaften and Hans L. Bodlaender for their careful proofreading.

References

[Ang80] ANGLUIN, D. Local and global properties in networks of processors. In *Symp. on Theory of Computing* (1980), pp. 82–93.

[Awe85] AWERBUCH, B. Complexity of network synchronization. *J. ACM* **32** (1985), 804–823.

[BB89] BEAME, P. W., AND BODLAENDER, H. L. Distributed computing on transitive networks: The torus. In *Symp. on Theoretical Aspects of Computer Science* (1989), B. Monien and R. Cori (Eds.), vol. 349 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 294–303.

[BN89] BOUABDALLAH, A., AND NAIMI, M. Parallel assignment to distinct identities in an arbitrary network. In *Third Annual Parallel Processing Symposium* (1989), H. Jelinek (Ed.).

[BT90] BODLAENDER, H. L., AND TEL, G. Bit-optimal election on synchronous rings. *Inf. Proc. Lett.* **36** (1990), 53–56.

[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.

[CS92] CIDON, I., AND SHAVITT, Y. Message terminate algorithms for anonymous rings of unknown size. In *6th Int. Workshop on Distributed Algorithms* (Haifa, 1992), A. Segall and S. Zaks (Eds.), vol. 647 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 264–276.

[DS80] DIJKSTRA, E. W., AND SCHOLTEN, C. S. Termination detection for diffusing computations. *Inf. Proc. Lett.* **11**, 1 (1980), 1–4.

[FLP85] FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. Impossibility of distributed consensus with one faulty process. *J. ACM* **32** (1985), 374–382.

[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.

[IJ93] ISRAELI, A., AND JALFON, M. Uniform self-stabilizing ring orientation. *Information and Computation* **104**, 2 (1993), 175–196.

[IR81] ITAI, A., AND RODEH, M. Symmetry breaking in distributive networks. In *Symp. on Theory of Computing* (1981), pp. 150–158.

[KG85] KORFHAGE, W., AND GAFNI, E. Orienting a unidirectional torus network. Manuscript, 1985.

[KK90a] KRANAKIS, E., AND KRIZANC, D. Computing boolean functions on anonymous hypercube networks. Report CS–R9040, Centre for Mathematics and Computer Science, Amsterdam, 1990.

[KK90b] KRANAKIS, E., AND KRIZANC, D. Computing boolean functions on Cayley networks. Report CS–R9061, Centre for Mathematics and Computer Science, Amsterdam, 1990.

[KMZ84] KORACH, E., MORAN, S., AND ZAKS, S. Tight upper and lower bounds for some distributed algorithms for a complete network of processors. In *3rd Symp. on Principles of Distributed Computing* (1984), pp. 199–207.

[KTZ88] KORACH, E., TEL, G., AND ZAKS, S. Optimal synchronization of ABD networks. In *CONCURRENCY '88* (Hamburg, 1988), F. H. Vogt (Ed.), vol. 335 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 353–367.

[LMW86] LOUI, M. C., MATSUSHITA, T. A., AND WEST, D. B. Election in a complete network with a sense of direction. *Inf. Proc. Lett.* **22** (1986), 185–187. *Addendum:* Inf. Proc. Lett. 28:327, 1988.

[LSP82] LAMPORT, L., SHOSTAK, R., AND PEASE, M. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.* **4** (1982), 382–401.

[MA89] MATIAS, Y., AND AFEK, Y. Simple and efficient election algorithms for anonymous networks. In *3rd Int. Workshop on Distributed Algorithms* (Nice, 1989), J.-C. Bermond and M. Raynal (Eds.), vol. 392 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 183–194.

[Mat89] MATTERN, F. *Verteilte Basisalgoritmen*, vol. 226 of *Informatik Fachberichte*. Springer-Verlag, Berlin, 1989 (285 p.).

[MW87] MORAN, S., AND WOLFSTAHL, Y. Extended impossibility results for asynchronous complete networks. *Inf. Proc. Lett.* **26** (1987), 145–151.

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

[PKR84] PACHL, J., KORACH, E., AND ROTEM, D. Lower bounds for distributed maximum finding algorithms. *J. ACM* **31** (1984), 905–918.

[RSB90] RAMANATHAN, P., SHIN, K. G., AND BUTLER, R. W. Fault-tolerant clock synchronization in distributed systems. *IEEE Computer* (1990), 33–42.

[San84] SANTORO, N. Sense of direction, topological awareness, and communication complexity. *ACM SIGACT News* **16** (1984), 50–56.

[SCP93] SYROTIUK, V. R., COLBOURN, C. J., AND PACHL, J. Wang tilings and distributed orientation on anonymous torus networks. In *7th Int. Workshop on Distributed Algorithms* (Les Diablerets, 1993), A. Schiper (Ed.), vol. 725 of *Lecture Notes in Computer Science*, Springer-Verlag.

[SP87] SYROTIUK, V., AND PACHL, J. A distributed ring orientation problem. In *2nd Int. Workshop on Distributed Algorithms* (Amsterdam, 1987), J. van Leeuwen (Ed.), vol. 312 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 332–336.

[Tel94] TEL, G. *Introduction to Distributed Algorithms*. Cambridge UP, 1994.

[TL86] TAN, R. B., AND LEEUWEN, J. VAN. General symmetric distributed termination detection. Tech. Rep. RUU–CS–86–2, Dept. of Computer Science, Univ. of Utrecht, The Netherlands, 1986.

[Vit85] VITÁNYI, P. M. B. Time-driven algorithms for distributed control. Tech. Rep. CS–R8510, Centre for Mathematics and Computer Science, Amsterdam, 1985.

Contents