Design and Analysis of Distributed Algorithms

Chapter 3:

ELECTION

Contents

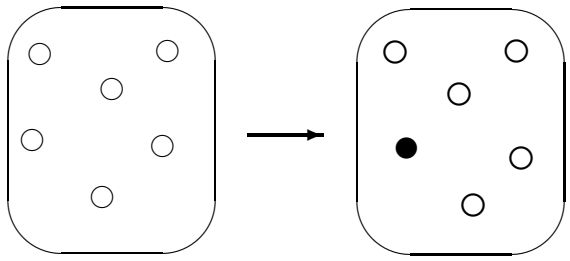


Figure 1: Electing a leader

1 Introduction

In a distributed environment, most applications often require a single entity to act temporarily as a central controller to coordinate the execution of a particular task by the entities. In some cases, the need for a single coordinator arises from the desire to simplify the design of the solution protocol for a rather complex problem; in other cases, the presence of a single coordinator is required by the nature of the problem itself.

The problem of choosing such a coordinator from a population of autonomous symmetric entities is known as *Leader Election* (**Elect**). Formally, the task consists in moving the system from an initial configuration where all entities are in the same state (usually called *available*), into a final configuration where all entities are in the same state (traditionally called *follower*), except one which is in a different state (traditionally called *leader*). There is no restriction on the number of entities that can start the computation, nor on which entity should become *leader*.

We can think of the Election problem as the problem of enforcing restriction *Unique Initiator* in a system where actually no such a restriction exists: the multiple initiators would first start the execution of an Election protocol; the sole *leader* will then be the unique initiator for the subsequent computation.

Since election provides a mechanism for breaking the symmetry among the entities in a distributed environment, it is at the basis of most control and coordination processes *(e.g., mutual exclusion, synchronization, concurrency control, etc.)* employed in distributed systems, and it is closely related to other basic computations *(e.g., minimum finding, spanning-tree construction, traversal).*

Impossibility Result

We will start considering this problem under the standard restrictions **R**: Bidirectional Links, Connectivity, and Total Reliability. There is unfortunately a very strong *impossibility* result about election.

Theorem 1.1 *Problem* **Elect** *is deterministically unsolvable under* **R**.

In other words, there is no deterministic protocol that will always correctly terminate within

finite time if the only restrictions are those in **R**.

To see why this is the case, consider a simple system composed of two entities, x and y, both initially *available* and with no different initial values; in other words, they are initially identical. If a solution protocol P exists, it must work under any conditions of message delays. Consider a *synchronous schedule* (i.e., an execution where communication delays are unitary) and let the two entities start the execution of P simultaneously. Since they are identical, they will execute the same rule, obtain the same result, compose and send (if any) the same message; thus, they will still be identical. If one of them receives a message, the other will receive the same message at the same time; the same computation will be performed by both, and so on. Their state will always be the same; hence if one becomes *leader*, so will the other. But this is against the requirement that there should be *only one* leader; in other words, P is *not* a solution protocol.

Additional Restrictions

The consequence of Theorem 1.1 is that, to break symmetry, we need additional restrictions and assumptions.

Some restrictions are not powerful enough. This is the case, for example, with the assumption that there is already available a spanning tree (i.e., restriction *Tree*). In fact, the two-nodes network in which we know election is impossible is a tree !

To determine which restrictions, added to **R**, will enable us to solve **Elect** we must consider the nature of the problem. The entities have an inherent behavioral symmetry: they all obey the same set of rules plus they have an initial state symmetry (by definition of election problem). To elect a leader means to break these symmetries; in fact, election is also called *symmetry breaking*. To be able to do so, from the start there must be something in the system that the entities can use, something that makes (at least one of) them different. Remember that any restriction limits the applicability of the protocol.

The most obvious restriction is *Unique Initiator* (UI): the unique initiator, knowing to be unique, becomes the *leader*. This is however "sweeping the problem under the carpet", saying that we can elect a leader if there is already a leader and it knows about it. The real problem is to elect a leader when many (possibly, all) entities are initiators; thus, without UI.

The restriction which is commonly used is a very powerful one, *Initial Distinct Values* (ID), which we have already employed to circumvent a similar impossibility result for constructing a spanning-tree with multiple initiators (in Section **??**). Distinct initial values are sometimes called *identifiers* or *ids* or *global names* and, as we will see, their presence will be sufficient to elect a leader; let $id(x)$ denote the distinct value of x. The use of this additional assumption is so frequent that the set of restrictions $\mathbf{IR} = \mathbf{R} \cup \{\text{ID}\}$ is called the *standard set for election*.

Solution Strategies

How can the difference in initial values be used to break the symmetry and elect a leader?

According to the election problem specifications, it does not matter which entity becomes leader. Using the fact the values are distinct, a possible strategy is to choose as a leader the entity with the *smallest* value; in other words, an election strategy is

STRATEGY *Elect Minimum*
(1) Find the smallest value;
(2) Elect as a leader the entity with that value.

IMPORTANT. Finding the minimum value is an important problem of its own, which we have already discussed for tree networks (Section **??**). Notice that, in that occasion, we found the minimum value *without* unique identifiers; it is the election problem that needs them !

A useful variant of this strategy is the one restricting the choice of the leader to the set of entities which initiate the protocol. That is,

STRATEGY *Elect Minimum Initiator*
(1) Find the smallest value among the initiators;
(2) Elect as a leader the entity with that value.

IMPORTANT. Notice that any solution implementing strategy *Elect Minimum* solves **Min** as well as **Elect**; not so the ones implementing *Elect Minimum Initiator*.

Similarly, we can define the *Elect Maximum* and the *Elect Maximum Initiator* strategies.

Another strategy is to use the distinct values to construct a rooted spanning tree of the network, and elect the root as the leader. In other words, an election strategy is

STRATEGY *Elect Root*
(1) Construct a rooted spanning-tree;
(2) Elect as the leader the root of the tree.

IMPORTANT. Constructing a (rooted) spanning-tree is an important problem of its own, which we have already discussed among the basic problems (Section **??**). Recall that **SPT**, like **Elect**, is *unsolvable* under **R**.

In the rest of this chapter we will examine how to use these strategies to solve **Elect** under election's standard set of restrictions $\mathbf{IR} = \mathbf{R} \cup \{\text{ID}\}$. We will do so by first examining special types of networks and then focusing on the development of topology-independent solutions.

2 Election in Trees

The tree is the connected graph with the 'sparsest' topology: $m = n - 1$.

We have already seen how to optimally find the smallest value using the saturation technique: protocol *MinF-Tree* in Section **??**. Hence strategy *Elect Minimum* leads to an election protocol where the number of messages in the worst case is

$$3n + k_\star - 4 \leq 4n - 4$$

Interestingly, also strategy *Elect Minimum Initiator* will have the same complexity (Exercise 9.1).

Consider now applying strategy *Elect Root*. Since the network *is* a tree, the only work required is to transform it into a *rooted* tree. It is not difficult to see how saturation can be used to solve the problem. In fact, if Full Saturation is applied, then a *saturated* node knows that itself and its parent are the only *saturated* nodes; furthermore, as a result of the saturation stage, every non-saturated entity has identified as its parent the neighbour closest to the saturated pair. In other words, saturation will root the tree not in a single node but in a pair of neighbours: the saturated ones.

Thus, to make the tree rooted in a single node we just need to choose only one of the two saturated nodes. In other words, the "election" among *all* the nodes is reduced to an "election" between the *two* saturated ones. This can be easily accomplished by having the *saturated* nodes communicate their identities and having the node with smallest identity become elected, while the other stays processing.

Thus, the *Tree_Election* protocol will be *Full Saturation* with the new rules and the routine *Resolve* shown in Fig. 2.

The number of message transmissions for the election algorithm *Tree_Election* will be exactly the same as the one experienced by full saturation with notification plus two "Election" messages; i.e., $\mathbf{M}[\textit{Tree\_Election}] = 3n + k_\star - 2 \leq 4n - 2$. In other words, it uses two more messages than the solution obtained using the strategy *Elect Minimum*.

Granularity of Analysis: Bit Complexity

From the discussion above, it would appear that the strategy *Elect Minimum* is "better" since it uses two fewer messages than strategy *Elect Root*. This assessment is indeed the only correct conclusion obtainable using the number of messages as the cost measure. Sometimes, this measure is too "coarse" and does not really allow us to see possibly important details; to get a more accurate picture, we need to analyze the costs at a "finer" level of granularity.

Let us re-examine the two strategies in terms of the number of *bits*. To do so, we have to distinguish between different types of messages since some contain counters and values, while others only a message identifier.

IMPORTANT. Messages that do not carry values but only a constant number of bits are called *signals* and in most practical systems they have significant less communication costs that *value messages*.

In *Elect Minimum*, only the n messages in the saturation stage carry a value, while all the

```
SATURATED
        Receiving(Election, id*)
        begin
            if id(x) < id* then
              become LEADER;
            else
              become FOLLOWER;
            endif
            send(``Termination'') to N(x) − {parent};
        end

PROCESSING
         Receiving(``Termination'')
         begin
              become FOLLOWER;
              send(``Termination'') to N(x) − {parent};
         end

Procedure Resolve
begin
     send(``Election'',id(x)) to parent;
     become SATURATED;
end
```

Figure 2: New rules and routine Resolved used for Tree_Election

others are signals; hence, the total number of *bits* transmitted will be

$$\mathbf{B}[Tree : Elect\_Min] = n\ (c\ + \log \mathbf{id}) + c\ (2n + \ k_\star - 2) \tag{1}$$

where **id** denotes the largest value sent in a message, and $c = O(1)$ denotes the number of bits required to distinguish among the different messages.

In *Elect Root*, only the "Election" message carries a node identity; thus the total number of *bits* transmitted is

$$\mathbf{B}[Tree : Elect\_Root] = 2\ (c\ + \log \mathbf{id}) + c\ (3n + \ k_\star - 2) \tag{2}$$

That is, in terms of number of bits, *Elect Root* is order of magnitude better than *Elect Minimum* ! In terms of signals and value messages, with *Elect Root* strategy we have only 2 value messages, with the *Elect Minimum* strategy we have n value messages.

Remember: measuring the number of bits gives us always a "picture" of the efficiency at a more refined level of granularity. Fortunately, it is not always necessary to go to such a level.

3 Election in Rings

We will now consider a network topology which plays a very important role in distributed computing: the *ring*, sometimes called *loop* network.

A ring consists of a single cycle of length n. In a ring, each entity has exactly two neighbours, (whose associated ports are) traditionally called *left* and *right* (see Figure 3).

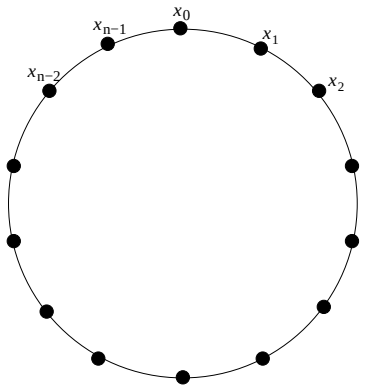


Figure 3: A ring network.

IMPORTANT. Note that the labelling might however be globally *inconsistent*; i.e., 'right' might not have the same meaning for all entities. We will return to this point later.

After trees, rings are the networks with the sparsest topology: $m = n$; however, unlike trees, rings have a complete structural symmetry (i.e., all nodes look the same) as opposed to the inherent asymmetry of trees (e.g., the existence of internal and leaf nodes).

We will denote the ring by $R = (x_0, x_1, , \ldots, x_{n-1})$. Let us consider the problem of electing a leader in a ring R, under the standard set of restriction for election, **IR**= {Bidirectional links, Connectivity, Total Reliability, Initial Distinct Values}, as well as the knowledge that the network is a ring (*Ring*). Denote by $id(x)$ the unique value associated to x.

Because of its structure, in a ring we will use almost exclusively the approach of minimum-finding as a tool for leader election. In fact we will consider both the *Elect Minimum* and the *Elect Minimum Initiator* approaches. Clearly the first solves both **Min** and **Elect**, while the latter solves only **Elect**.

NOTES. First notice that every protocol that elects a leader in a ring can be made to find the minimum value (if it has not already been determined) with an additional n message and time (Exercise 9.2). Further notice that, in the worst case, the two approaches coincide: all entities might be initiators.

Let us now examine how minimum-finding and election can be efficiently performed in a ring.

Since, in a ring each entity has only two neighbours, for brevity we will use the notation **other** to indicate $N(x)-$**sender** at an entity x.

3.1 All the Way

The first solution we will use is rather straightforward: When an entity starts, it will choose one of its two neighbours and send to it an "Election" message containing its id; an entity receiving somebody else's id, will send its id (if it has not already done so), and forward the received message along the ring (i.e., send it to its other neighbour) keeping track of the smallest id seen so far (including its own).

This process can be visualized as follows: (see Figure 4): each entity originates a message (containing its id), and this message travels "all the way" along the ring (forwarded by the other entities). Hence the name *All the way* we will use for the resulting protocol.

Each entity will eventually see everybody's else id (finite communication delays and total reliability ensure that) including the minimum value; it should thus be able to determine whether or not it is the (unique) minimum and thus the leader. When will this happen ? In other words,

Question. *When will an entity terminate its execution?*

Entities only forward messages carrying values other that their own: once the message with $id(x)$ arrives at x, it is no longer forwarded. Thus, every value will travel "all the way" along the ring only once. So, the communication activities will eventually terminate. But how does an entity know that the communication activities have terminated ? that no more messages will be arriving, and thus, the smallest value seen so far is really the minimum id ?

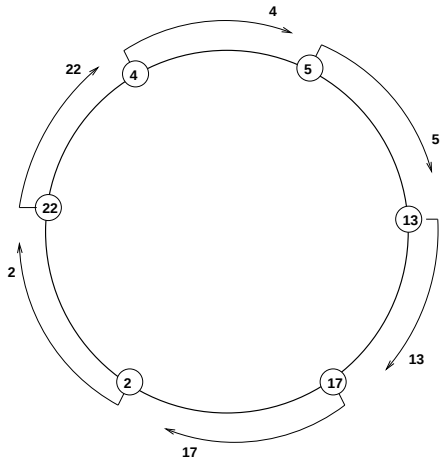


Figure 4: All the Way: every id travels all along the ring.

Consider a "reasonable" but unfortunately *incorrect* answer:

An entity knows that it has seen all values once it receives its value back.

The reasoning is that the message with its own id has to travel longer along the ring to reach x that those originated by the other entities; thus, these other messages will be received first. In other words, reception of its own message can be used to detect termination.

This reasoning is incorrect because it uses the (hidden) additional assumption that the system has FIFO communication channels; that is, the messages are delivered in the order they arrive. This restriction, called *Message Ordering*, is not part of election's standard set, few systems actually have it built-in, and the costs of offering it can be formidable.

So, whatever the answer, it must not assume FIFO channels. With this proviso, a "reasonable" but unfortunately still *incorrect* answer is the following:

An entity counts how many different values it receives; when the counter is equal to n, it knows it can terminate.

The problem is that this answer assumes that the entity knows n; but a priori knowledge of the ring size is *not* part of the standard restrictions for election. So it cannot be used.

It is indeed strange that termination should be difficult for such a simple protocol in such a clear setting. Fortunately, the last answer, although incorrect, provides us with the way out. In fact, although n is not known a priori, it can be *computed*. This is easily accomplished by having a counter in the Election message, initialized to 1 and incremented by each entity forwarding it; when an entity receives its id back, the value of the counter will be n.

Summarizing, we will use a counter at each entity, to keep track of how many different ids are received; and a counter in each message, so each entity can determine n. The protocol

PROTOCOL All the Way .

- States: $\mathcal{S} = \{$ASLEEP, AWAKE, FOLLOWER, LEADER$\}$;
  $\mathcal{S}_{INIT} = \{$ASLEEP$\}$;
  $\mathcal{S}_{TERM} = \{$FOLLOWER, LEADER$\}$.
- Restrictions: **RI** $\cup Ring$.

```
ASLEEP
      Spontaneously
      begin
           INITIALIZE;
           become AWAKE;
      end

      Receiving(‘‘Election’’, value, counter)
      begin
           INITIALIZE;
           send(‘‘Election’’, value, counter+1) to other;
           min:= Min{ min, value};
           count:= count+1;
           become AWAKE;
      end


AWAKE
     Receiving(‘‘Election’’, value, counter)
     begin
          if value ≠ id(x) then
            send(‘‘Election’’, value, counter+1) to other;
            min:= MIN{min,value};
            count:= count+1;
            if known then CHECK endif;
          else
            ringsize:= size*;
            known:= true;
            CHECK;
          endif
     end
```

Figure 5: Protocol *AlltheWay*

is shown in Figures 5 and 6.

The message originated by each entity will travel along the ring exactly once. Thus, there will be exactly n^2 messages in total, each carrying a counter and a value, for a total of $n^2 \log(\mathbf{id} + n)$ bits. The time costs will be at most $2n$ (Exercise 9.3). Summarizing,

$$\mathbf{M}[AlltheWay] = n^2 \tag{3}$$

$$\mathbf{T}[AlltheWay] \leq 2n - 1 \tag{4}$$

The solution protocol we have just designed is very expensive in terms of communication costs (in a network with 100 nodes it would cause $10,000$ message transmissions !).

The protocol can be obviously modified so to follow strategy *Elect Minimum Initiator*, finding

```
Procedure INITIALIZE
begin
     count:= 0;
     size:= 1;
     known:= false;
     send(‘‘Election’’, id(x), size) to right;
     min:= id(x);
end

Procedure CHECK
begin
     if count = ringsize then
       if min = id(x) then
         become LEADER;
       else
         become FOLLOWER;
       endif
     endif
end
```

Figure 6: Procedures of Protocol *All theWay*

the smallest value only among the initiators. In this case, those entities who do not initiate will not originate a message but just forward the others'. In this way, we would have fewer messages whenever there are fewer initiators.

In the modification we must be careful. In fact, in protocol *All the Way*, we were using an entity's own message to determine n so to be able to determine local termination. Now some entities will not have this information. This means that termination is again a problem. Fortunately, this problem has a simple solution requiring only n additional messages and time (Exercise 9.4). Summarizing, the costs of the modified protocol, *All the Way: Minit*, are as follows:

$$\mathbf{M}[AlltheWay : Minit] = nk_\star + n \tag{5}$$

$$\mathbf{T}[AlltheWay : Minit] \leq 3n - 1 \tag{6}$$

The modified protocol *All the Way: Minit* will in general use fewer message than the original one. In fact if only a constant number of entities initiate, it will uses only $O(n)$ messages, which is excellent. On the other hand, if *every* entity is an initiator, this protocol uses n messages *more* than the original one.

IMPORTANT. Notice that *All the Way* (in its original or modified version) can be used also in *unidirectional* rings with the same costs. In other words, it does not require the Bidirectional Links restriction. We will return to this point later.

3.2 As Far As It Can

To design an improved protocol, let us determine the drawback of the one we already have, *All the Way*. In this protocol, each message travels all along the ring.

Consider the situation (shown in Figure 7) of a message containing a large id, say 22 arriving at an entity x with a smaller id, say 4. In the existing protocol, x will forward this message, even though x knows that 22 is not the smallest value.

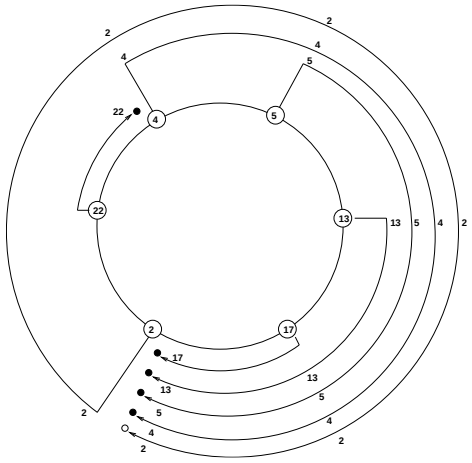


Figure 7: Message with a larger id do not need to be forwarded.

But our overall strategy is to determine the smallest id among all entities; if an entity determines that an id is not the minimum, there is no need whatsoever for the message containing such an id to continue travelling along the ring.

We will thus modify the original protocol *All the Way* so that an entity will only forward Election messages carrying an id *smaller* than the smallest seen so far by that entity. In other words, an entity will become an insurmountable obstacle for all messages with a larger id, "terminating" them.

Let us examine what happens with this simple modification. Each entity will originate a message (containing its id) which travels along the ring "as far as it can": until it returns to its originator or arrives at a node with a smaller id. Hence the name *AsFar* (As It Can) we will use for the resulting protocol.

Question. *When will an entity terminate its execution?*

The message with the smallest id will always be forwarded by the other entities; thus, it will travel all along the ring returning to its originator. The message containing another id will instead be unable to return to its originator as it will find an entity with a smaller id (and thus be terminated) along the way. In other words, only the message with the smallest id will return to its originator. This fact provides us with a termination detection mechanism.

If an entity receives a message with its own id, it knows that its id is the minimum, i.e. it is the leader; the other entities have all seen that message pass by (they forwarded it) but they still don't know that there will be no smaller ids coming by. Thus, to ensure their termination, the newly elected leader must *notify* them.

3.2.1 Message Cost

This protocol will definitely have fewer messages than the previous one. The exact number depends on several factors. Consider the cost caused by the *Election* message originated by x. This message will travel along the ring until it finds a smaller id (or complete the tour). Thus, the cost of its travel depends on how the ids are allocated on the ring. Also notice that what matters is whether an id is smaller or not than another, and not their actual value. In other words, what is important is the rank of the ids and how those are situated on the ring. Denote by $\#i$ the id whose rank is i.

Worst Case

Let us first consider the *worst possible case*. Id #1 will always travel all along the ring costing n messages. Id #2 will be stopped only by #1, so its cost in the worst case is $n-1$, achievable if #2 is located immediately after #1 in the direction it travels. In general, id $\#(i+1)$ will be stopped by any of those with smaller rank, and, thus, it will cost at most $n-i$ messages; this will happen if all those entities are next to each other, and $\#(i+1)$ is located immediately after them in the direction it will travel. In fact all the worst cases for each of the ids are *simultaneously* achieved when the ids are arranged in a (circular) order according to their rank and all messages are sent in the "increasing" direction (see Figure 9).

In this case, including also the n messages required for the final notification, the total cost will be

$$\mathbf{M}[AsFar] = n + \sum_{i=1}^{n} i = \frac{n\ (n+3)}{2} \tag{7}$$

That is, we will cut the number of the messages *at least in half*. From a theoretical point of view, the improvement is not significant; from a practical point of view, this is already a reasonable achievement. But we have so far analyze only the worst case. In general, the improvement will be much more significant. To see precisely how we need to perform a more detailed analysis of the protocol's performance.

IMPORTANT. Notice that *AsFar* can be used in *unidirectional*. In other words, it does not require the Bidirectional Links restriction. We will return to this point later.

The worst case gives us an indication of how "bad" things could get when the conditions are really bad. But how likely are such conditions to occur ? What costs can we generally expect ? To find out, we need to study the *average case* and determine the mean and the

PROTOCOL AsFar.

- States: $\mathcal{S}$ = {ASLEEP, AWAKE, FOLLOWER, LEADER};
  $\mathcal{S}_{INIT}$ = {ASLEEP};
  $\mathcal{S}_{TERM}$ = {FOLLOWER, LEADER}.
- Restrictions: **RI** $\cup$*Ring*.

```
ASLEEP
      Spontaneously
      begin
           INITIALIZE;
           become AWAKE;
      end

      Receiving(‘‘Election’’, value)
      begin
           INITIALIZE;
           if value < min then
             send(‘‘Election’’, value) to other;
             min:= value;
           endif
           become AWAKE;
      end


AWAKE
     Receiving(‘‘Election’’, value)
     begin
          if value < min then
            send(‘‘Election’’, value) to other;
            min:= value;
          else
            if value min then NOTIFY endif;
          endif
     end

     Receiving(Notify)
          send(Notify) to other;
          become FOLLOWER;
     end
```

where the procedures *Initialize* and *Notify* are as follows:

```
Procedure INITIALIZE
begin
     send(‘‘Election’’, id(x)) to right;
     min:= id(x);
end

Procedure NOTIFY
begin
     send(Notify) to right;
     become LEADER;
end
```

Figure 8: Protocol *AsFar*

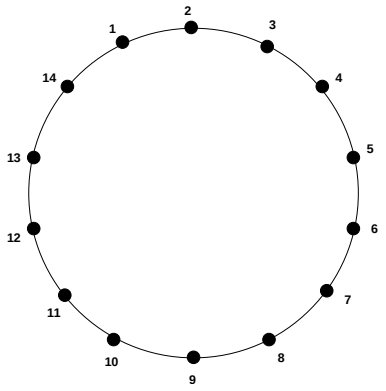


Figure 9: Worst case setting for protocol *AsFar*

variance of the cost of the protocol.

Average Case: Oriented Ring

We will first consider the case when the ring is *oriented*; that is, "right" means the same to all entities. In this case, all messages will travel in only one direction, say clockwise.

IMPORTANT. Because of the unique nature of the ring network, this case *coincides* with the execution of the protocol in a *unidirectional* ring. Thus, the results we will obtain will hold for those rings.

To determine the average case behavior, we consider all possible arrangements of the ranks $1, \ldots, n$ in the ring as equally likely. Given a set of size a, we denote by $C(a, b)$ the number of subsets of size b that can be formed from it.

Consider the id $\#i$ with rank i; it will travel clockwise exactly k steps if and only if the ids of its $k-1$ clockwise neighbours are larger than it (and thus will forward it), while the id of its k-th clockwise neighbour is smaller (and thus will terminate it).

There are $i-1$ ids smaller than $\#i$ from which to choose those $k-1$ smaller clockwise neighbours, and there are $n-i$ ids larger than $\#i$ from which to choose the k-th clockwise neighbour. In other words, the number of situations where $\#i$ will travel clockwise exactly k steps is $C(i-1, k-1)C(n-i, 1)$, out of the total number of $C(n-1, k-1)C(n-k, 1)$ possible situations.

Thus, the probability $P(i, k)$ that $\#i$ it will travel clockwise exactly k steps is

$$P(i,k) = \frac{C(i-1,k-1)C(n-i,1)}{C(n-1,k-1)C(n-k,1)} \tag{8}$$

The smallest id, #1, will travel the full length n of the ring. The id $\#i$, $i > 1$, will travel less; the expected distance will be

$$E_i = \sum_{k=1}^{n-1} k \ P(i,k) \tag{9}$$

Therefore, the overall expected number of message transmissions is

$$E = n + \sum_{i=1}^{n-1} \sum_{k=1}^{n-1} k \ P(i,k) = n + \sum_{k=1}^{n-1} \frac{n}{k+1} = nH_n \tag{10}$$

where $H_n = 1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n}$ is the *n-th Harmonic* number.

To obtain a close formula, we use the fact that the function $f(x) = \frac{1}{x}$ is continuous, linear and decreasing; thus $\int_1^\infty \frac{1}{x}\, dx = \lim_{n\to\infty} \int_1^n \frac{1}{x}\, dx = \lim_{n\to\infty} \ln x \Big|_1^n = \lim_{n\to\infty}(\ln n - \ln 1 + c) = \ln n + c$. Hence, $H_n = \ln n + O(1) \approx .69 \log n + O(1)$; thus

Theorem 3.1
In oriented and in unidirectional rings, protocol AsFar *will cost* $nH_n \approx .69n \log n + O(n)$ *messages on the average.*

This is indeed great news: on the average, the message cost is order of magnitude less than in the worst case. For $n = 1,024$, this means that on the average we have $7,066$ messages instead of $525,824$, a considerable difference !

If we use the strategy of electing the *minimum initiator* instead, we obtain the same bound but as a function of the number k_* of initiators:

Theorem 3.2
In oriented and in unidirectional rings, protocol AsFar-Minit *will cost* $nH_{k_\star} \approx .69n \log k_\star$ *messages on the average.*

Average Case: Unoriented Ring

Let consider now what will happen on the average in the general case, when the ring is *unoriented*. As before, we consider all possible arrangements of the ranks $1, \ldots, n$ of the values in the ring as equally likely. The fact that the ring is not oriented means that when two entities send a message to their "right" neighbours, they might send it in different directions.

Let us assume that at each entity the probability that "right" coincides with the clockwise direction is $\frac{1}{2}$. Alternatively, assume that an entity, as its first step in the protocol, flips a fair coin (i.e., probability $\frac{1}{2}$) to decide the direction it will use to send its value. We shall call the resulting probabilistic protocol *ProbAsFar*.

Theorem 3.3
In unoriented rings, Protocol ProbAsFar *will cost* $\frac{\sqrt{(2)}}{2} n H_n \approx .49 n \log n$ *messages on the average.*

A similar bound holds if we use the strategy of electing the *minimum initiator*:

Theorem 3.4
In unoriented rings, protocol ProbAsFar-Minit *will cost* $\frac{\sqrt{(2)}}{2} n H_{k_\star} \approx .49 n \log k_\star$ *messages on the average.*

What is very interesting about the bound expressed by Theorem 3.3 is that it is *better* (i.e., smaller) than the one expressed by Theorem 3.1. The difference between the two bounds is restricted to the constant and is rather limited. In numerical terms, the difference is not outstanding: $5,018$ instead of $7,066$ messages on the average when $n = 1,024$.

In practical terms, from the algorithm design point of view, it indicates that we should try to have the entities send their initial message in different directions (as in the probabilistic protocol) and not all in the same one (like in the oriented case). To simulate the initial "random" direction, different means can be used. For example, each entity x can choose (its own) "right" if $id(x)$ is even, (its own) "left" otherwise.

This result has also a theoretical relevance that will become apparent later, when we will discuss lower bounds, and will have a closer look to the nature of the difference between oriented and unoriented rings.

3.2.2 Time Costs

The time costs are the same as the ones of *All the Way* plus an additional $n - 1$ for the notification. This can however be halved by exploiting the fact that the links are bidirectional, and broadcasting the notification; this will require an extra message but halve the time.

3.2.3 Summary

The main drawback of protocol *AsFar* is that there still exists the possibility that a very large number of messages ($O(n^2)$) will be exchanged. As we have seen, on the average, the use of the protocol will cost only $O(n \log n)$ messages. There is however no guarantee that this will happen the next time the protocol will be used. To give such a guarantee, a protocol must have a $O(n \log n)$ worst case complexity.

3.3 Controlled Distance

We will now design a protocol which has a *guaranteed* $O(n \log n)$ message performance.

To achieve this goal, we must first of all determine what causes the previous protocol to use $O(n^2)$ messages, and then find ways around it.

The first thing to observe is that in *AsFar* (as well as in *All the Way*), an entity makes only one attempt to become leader and does so by originating a message containing its id. Next observe that, once this message has been created and sent, the entity has no longer control over it: in *All the Way* the message will travel all along the ring; in *AsFar* it will be stopped if it finds a smaller id.

Consider now the situation that causes the worst case for protocol *AsFar*; this is when the ids are arranged in increasing order along the ring, and all entities identify "right" with the clockwise direction (see Figure 9). The entity x with id 2 will originate a message which will causes $n-2$ transmissions. When x receives the message containing id 1, x finds out that its own value is not the smallest, and thus its message is destined to be wasted. However, x has no means to stop it, since has no longer control over that message.

Let us take these observations into account to design a more efficient protocol. The key design goal will be to make an entity retain some control over the message it originates. We will use several ideas to achieve this:

(1) *limited distance:*
the entity will impose a *limit* on the distance its message will travel; in this way, the message with id 2 will not travel "as far as it can" (i.e., at distance $n-2$) but only up to some pre-defined length.

(2) *return* (or *feedback*) *messages:*
if, during this limited travel, the message is *not* terminated by an entity with smaller id, it will *return* back to its originator to get authorization for further travel; in this way, if the entity with id 2 has seen id 1, it will abort any further travel of its own message.

Summarizing, an entity x will originate a message with its own id, and this message will travel until it is terminated or it reaches a certain distance dis; if it is not terminated, the message returns to the entity. When it arrives, x knows that, on this side of the ring there are no smaller ids within the travelled distance dis.

The entity must now decide if to allow its message to travel a further distance; it will do so only if it knows for sure that there are no smaller ids within distance dis on the other side of the ring as well. This can be achieved as follows:

(3) *check both sides:*
the entity will send a message in both directions; only if they both return, they will be allowed to travel to a further distance.

As a consequence, instead of a single global attempt at leadership, an entity will go through several attempts, which we shall call *Electoral Stages*: an entity enters the next stage only if it passes the current one (i.e., both messages return). See Figure 11. If an entity is *defeated* in an electoral stage (i.e., at least one of its messages does not return), it still will have to continue its participation in the algorithm forwarding the messages of those entities that are still undefeated.

Although the protocol is almost all outlined, some fundamental issues are still unresolved. In particular, the fact that we now have several stages can have strange consequences in the execution.

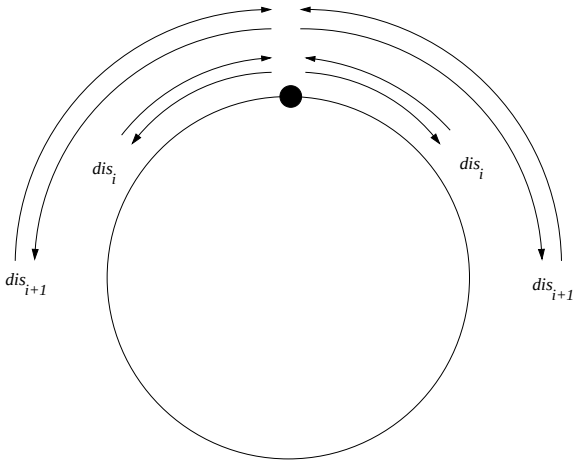


Figure 10: Controlled Distances: A message (in bold) travels no more than *dis(i)*; if it is not discarded, a feedback is sent back to the originator. A *candidate* that receives a feedback from both sides starts the next stage.

IMPORTANT. Because of variations in communication delays, it is possible that, at the same time instant, entities in different parts of the ring are in different electoral stages. Furthermore, since we are only using the standard restrictions for elections, messages can be delivered out of order; thus, it might be possible that messages from a higher stage will arrive at an entity before the ones from the current one.

We said that an entity is defeated if it does not receive one of its messages back. Consider now an entity x; it has sent its two messages and it is now waiting to know the outcome. Let us say that one of its messages has returned but the other not yet. It is possible that the message is very slow coming (e.g., experiencing long transmission delays) or that it is not coming at all (i.e., it found a smaller id on the way). How can x know ? How long will x have to wait before taking a decision ? (a decision *must* be taken within finite time). More specifically, what will x do if, in the meanwhile, it receives a message from a higher stage ? The answer to all these questions is fortunately simple:

(4) *the smallest wins:*
If, at any time, a candidate entity receives a message with a *smaller* id, it will become *defeated*, regardless of the stage number.

Notice that this creates a new situation: a message returns to its originator and finds it *defeated*; in this case, the message will be terminated.

The final issue we need to address is termination. The limit to the travel distance for a message in a given stage will depend on the stage itself; let dis_i denote the limit in stage i. Clearly, these distances must be monotonically increasing; i.e., $dis_i > dis_{i-1}$. The messages from an entity whose id is not the minimum will sooner or later encounter a smaller id in

their travel, and not return to their originator.
Consider now the entity s with the smallest id. In each stage, both its messages will travel the full allocated distance (since no entity can terminate them) and return, making s enter the next stage. This process will continue until $dis_i \geq n$; at this time, each message will complete a full tour of the ring reaching s *from the other side* ! When this happens, s will know that it has the smallest value and, thus, it is the leader. It will then start a notification process so all the other entities can enter a terminal state.

A synthetic description of the protocol will thus be:

- In each electoral stage there are some *candidates*;
- each *candidate* sends a message in both directions carrying its own id (as well as the stage number);
- a message travels until it encounters a smaller id or it reaches a certain distance (whose value depends on the stage);
- if a message does not encounter a smaller id, it will return back to its originator;
- a *candidate* which receives both its own messages back survives this stage and starts the next one.

with three *meta* rules:

- If a *candidate* receives its message from the opposite side it sent it, it becomes the *leader* and notifies all other entities of termination.
- If a *candidate* receives a message with a smaller id, it becomes *defeated*, regardless of the stage number.
- a *defeated* entity forwards the messages originating from other entities; if the message is notification of termination, it will terminate.

The fully specified protocol *Control* is shown in Figures 11 and 12, where dis is a monotonically increasing function.

3.3.1 Correctness.

The correctness of the algorithm follows from the dynamics of the rules: the messages containing the smallest id will always travel all the allocated distance, and every entity still *candidate* they encounter will be transformed in *defeated*; the distance is monotonically increasing in the number of stages; hence, eventually, the distance will be at least n. When this happens, the messages with the smallest value will travel all along the ring; as a result, their originator becomes *leader* and all the others are already *defeated*.

PROTOCOL Control.

- States: $\mathcal{S}$ = {ASLEEP, CANDIDATE, DEFEATED, FOLLOWER, LEADER};
  $\mathcal{S}_{INIT}$ = {ASLEEP};
  $\mathcal{S}_{TERM}$ = {FOLLOWER, LEADER}.
- Restrictions: **RI** $\cup$*Ring*.

```
ASLEEP
      Spontaneously
      begin
           INITIALIZE;
           become CANDIDATE;
      end

      Receiving(``Forth'', id*, stage*, limit*)
      begin
           if id* < id(x) then
             PROCESS-MESSAGE;
             become DEFEATED
           else
             INITIALIZE;
             become CANDIDATE;
           endif
      end


CANDIDATE
        Receiving(``Forth'', id*, stage*, limit*)
        begin
             if id* < id(x) then
               PROCESS-MESSAGE;
               become DEFEATED
             else
               if id* = id(x) then NOTIFY endif;
             endif
        end

        Receiving(``Back'', id*)
        begin
             if id* = id(x) then CHECK endif;
        end

        Receiving(Notify)
        begin
             send(Notify) to other;
             become FOLLOWER;
        end


DEFEATED
       Receiving(⋆)
       begin
            send(⋆) to other;
            if ⋆ = Notify then become FOLLOWER endif;
       end
```

Figure 11: Protocol *Control*

```
Procedure INITIALIZE
begin
     stage:= 1;
     limit:= dis(stage);
     count:= 0;
     send(``Forth'', id(x), stage, limit) to N(x);
end

Procedure PROCESS-MESSAGE
begin
     limit*:=limit*-1;
     if limit* =0 then
       send(``Back'',id*, stage*) to sender;
     else
       send(``Forth'', id*, stage*, limit*) to other;
     endif
end

Procedure CHECK
begin
     count:=count+1;
     if count = 1 then
       count:= 0
       stage:= stage+1
       limit:= dis(stage);
       send(``Forth'', id(x), stage, limit) to N(x);
     endif
end

Procedure NOTIFY
begin
     send(Notify) to right;
     become LEADER;
end
```

Figure 12: Procedures used by protocol *Control*

3.3.2 Costs.

The costs of the algorithm depend totally on the choice of the function dis used to determine the maximum distance a "Forth" message can travel in a stage.

Messages.
If we examine the execution of the protocol at some global time t, because communication delays are unpredictable, we can find not only that entities in different parts of the ring are in different states (which is expected), but also that entities in the *candidate* state are in different stages. Moreover, because there is no message ordering, messages from high stages (the "future") might overtake messages and arrive at an entity still in a lower stage (the "past").

Still, we can visualize the execution as proceeding in *logical* stages; it is just that different entities might be executing the same stage at different times.

Focus on stage $i > 1$, and consider the entities that will start this stage; these n_i entities are those who survived stage $i - 1$.

To survive stage $i - 1$, the id of x must be smaller than the ids of its neighbours at distance up to $dis(i)$ on each side of the ring. Thus, within any group of $dis(i) + 1$ consecutive entities, at most one can survive stage $i - 1$ and start stage i. In other words,

$$n_i \leq \lfloor \frac{n}{dis(i-1)+1} \rfloor \tag{11}$$

An entity starting stage i will send "Forth" messages in both directions; each message will travel at most $dis(i)$, for a total of $2n_i\ dis(i)$ message transmissions.

Let us examine now the "Back" messages. Each entity that survives this stage will receive such a message from both sides; since n_{i+1} entities survive this stage, this gives an additional $2n_{i+1}dis(i)$ messages. Each entity which started but did *not* survive stage i will receive either no or at most one "Back" message, causing a cost of *at most* $dis(i)$; since there are $n_i - n_{i+1}$ such entities, they will cost no more than an additional $(n_i - n_{i+1})dis(i)$ messages in total. So, in total, the transmissions for "Back" messages are at most $2n_{i+1}dis(i)+(n_i-n_{i+1})dis(i)$.

Summarizing, the total number of messages sent in stage $i > 1$ will be no more than

$$2\ n_i\ dis(i) + 2\ n_{i+1}\ dis(i) + (n_i - n_{i+1})\ dis(i) = (3\ n_i + n_{i+1})\ dis(i) \leq$$
$$(3\ \lfloor \tfrac{n}{dis(i-1)+1} \rfloor + \lfloor \tfrac{n}{dis(i)+1} \rfloor)\ dis(i) < n\ (3\ \tfrac{dis(i)}{dis(i-1)} + 1)$$

The first stage is a bit different, as every entity starts; the n_2 entities which survive this stage will have caused the messages carrying their id to travel to distance $dis(1)$ and back on both sides, for a total of $4n_2\ dis(1)$ messages. The $n - n_2$ that will not survive will cause at most three messages each (two "Forth" and one "Back") to travel at distance $dis(1)$, for a total of $3(n_1 - n_2)dis(1)$ messages. Hence the first stage will cost no more than

$$(3n + n_2)\ dis(1) \leq (3n + \tfrac{n}{dis(1)+1})\ dis(1) < n\ (3\ dis(1) + 1)$$

To determine the total number of messages we then need to know the total number k of stages. We know that a leader is elected as soon as the message with the smallest value makes a complete tour of the ring; that is, as soon as $dis(i)$ is greater or equal to n. In other words, k is the smallest integer such that $dis(k) \geq n$; such an integer is called the *pseudo-inverse* of n and denoted by $dis^{-1}(n)$.

So, the total number of messages used by protocol *Control* will be at most

$$\mathbf{M}[Control] \leq n \sum_{i=1}^{dis^{-1}(n)} (3 \frac{dis(i)}{dis(i-1)} + 1) + n \tag{12}$$

where $dis(0) = 1$ and the last n messages are those for the final notification.

To really finalize the design, we must choose the function dis. Different choices will results in different performances.

Consider, for example, the choice $dis(i) = 2^{i-1}$; then $\frac{dis(i)}{dis(i-1)} = 2$ (i.e., we double the distance every time) and $dis^{-1}(n) = \lceil \log n \rceil + 1$, which in Expression 12 yields

$$\mathbf{M}[Control] \leq 7\ n \log n + O(n)$$

which is what we were aiming for: a $O(n \log n)$ worst case !

The constant can be however further improved by carefully selecting dis. Determining which function is best is rather difficult. Let us restrict the choice to among the functions where, like the one above, the ratio between consecutive values is constant; i.e., $\frac{dis(i)}{dis(i-1)} = c$. For these functions, $dis^{-1}(n) = \lceil log_c(n) \rceil + 1$; thus, Expression 12 becomes

$$\frac{3c+1}{\log c} n \log n + O(n)$$

Thus, with all of them, protocol *Control* has a guaranteed $O(n \log n)$ performance !

The "best" among those functions will be the one where $\frac{3c+1}{\log c}$ is minimized; since distances must be integer quantities, also c must be an integer. Thus the best such choice is $c = 3$ for which we obtain

$$\mathbf{M}[Control] \leq 6.309\ n \log n + O(n) \tag{13}$$

Time.
The ideal time complexity of procedure *Control* is easy to determine; the time required by stage i is the time needed by the message containing the smallest id to reach its assigned distance and come back to its originator; hence exactly $2dis(i)$ time units. An additional n time units are needed for the final notification, as well as for the initial wake-up of the entity with the smallest id. This means that the total time costs will be at most

$$\mathbf{T}[Control] \leq 2n + \sum_{i=1}^{dis^{-1}(n)} 2\ dis(i) \tag{14}$$

Again, the choice of dis will influence the complexity. Using any function of the form $dis(i) = c^{i-1}$, where c is a positive integer will yield $O(n)$ time. The determination of the best choice from the time costs point of view is left as an exercise. (Exercise)

3.3.3 $\star$ Electing Minimum Initiator

Let us use the strategy of electing a leader only among the initiators. Denote as usual by $k_\star$ the number of initiator. Let us analyze the worst case.

In the analysis of protocol *Control*, we have seen that those that survive stage i contribute $4\ dis(i)$ messages each to the cost, while those that do not survive contribute at most $3dis(i)$ messages each. This is still true in the modified version *Control-Minit*; what changes, is the values of the number n_i of entities that will start that stage. Initially, $n_1 = k_\star$. In the worst case, the $k_\star$ initiators are placed far enough from each other in the ring that each completes the stage without interfering with the others; if the distances between them are large enough, each can continue to go to higher stages without coming into contact with the others, thus causing $4\ dis(i)$ messages.

For how many stages can this occur ? This can occur as long as $dis(i) < \frac{n}{k_\star+1}$. That is, in the worst case, $n_i = k_\star$ in each of the first $l = dis^{-1}(\frac{n}{k_\star+1} - 1)$ stages, and the cost will be $4\ k_\star dis(i)$ messages. In the following stages instead, the initiators will start interfering with each other, and the number of survivors will follow the pattern of the general algorithm: $n_i \leq \lfloor \frac{n_1}{dis(i-1)+1} \rfloor$.

Thus, the total number $\mathbf{M}[Control\text{-}Minit]$ of messages in the worst case will be at most

$$\mathbf{M}[Control - Minit] \leq 4\ k_\star \sum_{i=1}^{l} dis(i)\ +\ n \sum_{i=l+1}^{dis^{-1}(n)} (3\ \frac{dis(i)}{dis(i-1)} + 1)\ +\ n \qquad (15)$$

3.4 Electoral Stages

In the previous protocol we have introduced and used the idea of *limiting the distances* to control the complexity of the original "as far as it can" approach. This idea requires that an entity makes several successive attempts (at increasing distances) to become a leader.

The idea of not making a single attempt to become a leader (as it was done in *All the Way* and in *AsFar*), but instead of proceeding in stages is a very powerful algorithmic tool of its own. It allows us to view the election as a sequence of *electoral stages* : at the beginning of each stage, the "candidates" run for election; at the end of the stage, some "candidates" will be defeated, the others will start the next stage. Recall that "stage" is a logical notion, and it does not requires the system to be synchronized; in fact, parts of the system may run very fast while other parts may be slow in their operation, so different entities might execute a stage at totally different times.

We will now see how the proper use of this tool allows us to achieve even better results, and *without* controlling the distances and *without* return (or feedback) messages.

To simplify the presentation and the discussion, we will temporarily assume that there is *Message Ordering* (i.e., the links are FIFO); we will remove the restriction immediately after.

As before, we will have each *candidate* send a message carrying its own id in both directions. Without setting an a priori fixed limit on the distance these messages can travel, we still would like to avoid them to travel unnecessarily far (costing too many transmissions). The strategy to achieve this is simple and effective:

a message will travel until it reaches another candidate

that is in the same (or higher) stage. The consequence of this simple strategy is that, in each stage,

a candidate will receive a message from each side

thus, it will know the ids of the neighbouring candidate on each side. We will use this fact to decide whether a candidate x enters the next stage: x will survive this stage only if the two received ids are not smaller than its own $id(x)$ (recall we are electing the entity with smallest id); otherwise, it becomes *defeated*. As before, we will have *defeated* entities continue to participate by forwarding received messages.

Correctness and termination are easy to verify. Observe that the initiator with smallest identity will never become *defeated*; on the other hand, at each stage, its message will transform into *defeated* the neighbouring *candidate* on each side (regardless of their distance). Hence, the number of *candidates* decreases at each stage. This means that, eventually, the only *candidate* left is the one with the minimum id. When this happens, its messages will travel all along the ring (forwarded by the *defeated* entities), and reach it. Thus, a *candidate* receiving its own messages back knows that all other entities are *defeated*; it will then become *leader* and notify all other entities of termination.

Summarizing (see also Fig. 13):

- A candidate x sends a message in both directions carrying its identity; these messages will travel until they encounter another candidate node.
- By symmetry, entity x will receive two messages, one from the "left" and one from the "right" (independently of any sense of direction); it will then become *defeated* if at least one of them carries an identity smaller that its own; if the received identities are both larger than its own, it starts the next stage; finally, if the received identities are its own, it becomes *leader* and notify all entities of termination.
- A *defeated* node will forward any received election message, and each non-initiator will automatically become defeated upon receiving an election message.

The protocol is shown in Figure 14, where **close** and **open** denote the operation of closing a port (with the effect of enqueueing incoming messages), and opening a closed port (dequeueing the messages), respectively, and where procedure Initialize is shown in Fig. 15.

$x > Min\{y, z\} \Rightarrow x$ *defeated*

$x < Min\{y, z\} \Rightarrow x$ *candidate* next stage

$x = Min\{x, y\} \Rightarrow x$ *leader*

Figure 13: A *candidate* x in an electoral stage.

3.4.1 Messages

It is not so obvious that this strategy is more efficient than the previous.

Let us first determine the number of message exchanged during a stage. Consider the segment of the ring between two neighbouring *candidate* in stage i, x and $y = r(i, x)$; in this stage x will send a message to y and y will send one to x. No other messages will be transmitted during this stage in that segment. In other words, on each link, only *two* messages will be transmitted (one in each direction) in this stage. Therefore, in total, $2n$ message exchanges will be performed during each stage.

Let us determine now the number of stages. Consider a node x that is *candidate* at the beginning of stage i and is not defeated during this stage; let $y = r(i, x)$ and $z = l(i, x)$ be the first entity to the right and to the left of x, respectively, that are also *candidate* in stage i (Figure 16).

It is not difficult to see that, if x survives stage i, both $r(i, x)$ and $l(i, x)$ will be *defeated*. Therefore, at least half of the candidates are *defeated* at each stage; In other words, at most half of them survive:

$$n_i \leq \frac{n_{i-1}}{2}$$

Since $n_1 = n$, the total number of stages σ_{Stages} is at most $\sigma_{Stages} \leq \lceil \log n \rceil + 1$.

Combining the two observations we obtain

$$\mathbf{M}[Stages] \leq 2\ n \log n + O(n) \tag{16}$$

That is, protocol *Stages* outperforms protocol *Control*.

Observe that this bound is achievable in practice (Exercise 9.9). Further note that, if we use the *minimum initiator* approach the bound will become

$$\mathbf{M}[Stages] \leq 2\ n \log k_* + O(n) \tag{17}$$

PROTOCOL Stages.

- States: $\mathcal{S}$ = {ASLEEP, CANDIDATE, WAITING, DEFEATED, FOLLOWER, LEADER}; $\mathcal{S}_{INIT}$ = {ASLEEP}; $\mathcal{S}_{TERM}$ = {FOLLOWER, LEADER}.
- Restrictions: **RI** $\cup Ring$.

```
ASLEEP
      Spontaneously
      begin
           INITIALIZE;
           become CANDIDATE;
      end

      Receiving(``Election'', id*, stage*)
      begin
           INITIALIZE;
           min:= Min(id*,min);
           close(sender);
           become WAITING;
      end


CANDIDATE
        Receiving(``Election'', id*, stage*)
        begin
             if id* ≠ id(x) then
               min:= Min(id*,min);
               close(sender);
               become WAITING;
             else
               send(Notify) to N(x);
               become LEADER;
        end

WAITING
       Receiving(``Election'', id*, stage*)
             open(other);
             stage:= stage+1;
             min:= Min(id*,min);
             if min= id(x) then
               send(``Election'', id(x), stage) to N(x);
               become CANDIDATE;
             else
               become DEFEATED;
             endif
       end

DEFEATED
        Receiving(⋆)
        begin
             send(⋆) to other;
             if ⋆ = Notify then become FOLLOWER endif;
        end
```

Figure 14: Protocol *Stages*

```
Procedure INITIALIZE
begin
     stage:= 1;
     count:= 0;
     min:= id(x);
     send(''Election'', id(x), stage) to N(x);
end
```

Figure 15: Procedure Initialize used by protocol *Stages*

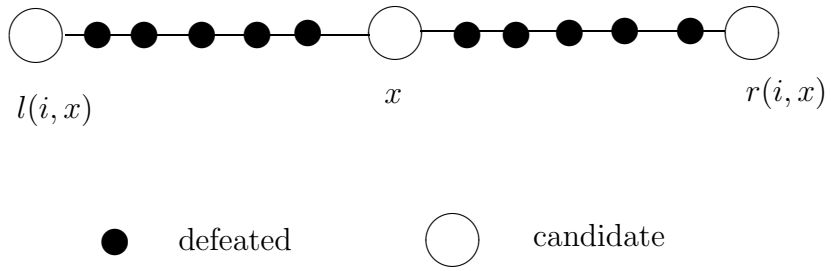


Figure 16: If x survives this stage, its neighbouring *candidates* will not.

3.4.2 Removing Message Ordering

The correctness and termination of *Stages* are easy to follow also because we have assumed in our protocol that there is *Message Ordering*. This assumption ensured that the two messages received by a *candidate* in stage i are originated by *candidates* also in stage i. If we remove the *Message Ordering* restriction, it is possible that messages arrive out of order, and that a message sent in stage $j > i$ arrives *before* a message sent in stage i.

Simple Approach
The simplest way to approach this problem is by enforcing the "effects" of message ordering, without really having it.
1. First of all, each message will also carry the stage number of the entity originating it.
2. When a *candidate* node x in stage i receives a message $M*$ with stage $j > i$ it will not process it but locally enqueue it, until it has received from that side (and processed) all the messages from stages $i, i+1, \ldots, j-1$, which have been "jumped over" by $M*$; it will then process $M*$.

The only modification to protocol *Stages* as described in Fig. 14, is the addition of the local enqueueing of messages (Exercise 9.6); since this is only local processing, the message and time costs are unchanged.

*Stages**

An alternative approach is to keep track of a message "jumping over" others but without enqueuing it locally. We shall describe it in some details and call *Stages** the corresponding protocol.

1. First of all, we will give a stage number to all nodes: for a *candidate* entity, it is the current stage; for a *defeated* entity, it is the stage in which it was defeated. We will then have a *defeated* node forward only messages from higher stages.

2. A *candidate* node x in stage i receiving an Election message $M*$ with stage $j > i$ will use the id included in the message, $id*$, and make a decision about the outcome of the stage i as if they both were in the same stage.
- If x is defeated in this round, then it will forward the message $M*$.
- If x survives, it means that $id(x)$ is smaller not only of $id*$ in $M*$ but also of the ids in the messages "jumped over" by $M*$ (Exercise 9.13). In this case, x can act as it has received already from that side all the messages from stages $i, i+1, \ldots, j$, and they all have an id larger than $id(x)$. We will indicate this fact by saying that x has now a *credit* of $j - i$ messages on that port. In other words, if a *candidate* x has a *credit* $c > 0$ associated with a port, it does not have to wait for a message from that port during the current stage. Clearly, the credit must be decreased in each stage.

To write the set of rules for protocol *Stages** is a task that, although not difficult, requires great care and attention to details (Exercise 9.12); similar characteristics has the task of proving the correctness of the protocol *Stages** (Exercise 9.14).

As for the resulting communication complexity, the number of messages is never more (sometimes less) than with message ordering (Exercise 9.15).

Interestingly, if we attempt to measure the *ideal time* complexity, we will only see executions with message ordering. In other words,

the phenomenon of messages delivered out of order will disappear !

This is yet another case showing how biased and limited (and thus dangerous) *ideal time* is as a cost measure.

3.5 Stages with Feedback

We have seen how, with the proper use of electoral stages in protocol *Stages*, we can obtain a $O(n \log n)$ performance without the need of controlling the distance travelled by a message.

In addition to controlled distances, protocol *Control* uses also a "feedback" technique: if a message successfully reaches its target, it returns back to its originator, providing it with a "positive feedback" on the situation it has encountered. Such a technique is missing in *Stages*: a message always successfully reaches its target (the next candidate in the direction it travels) which could be at an unpredictable distance; however the use of the message ends there.

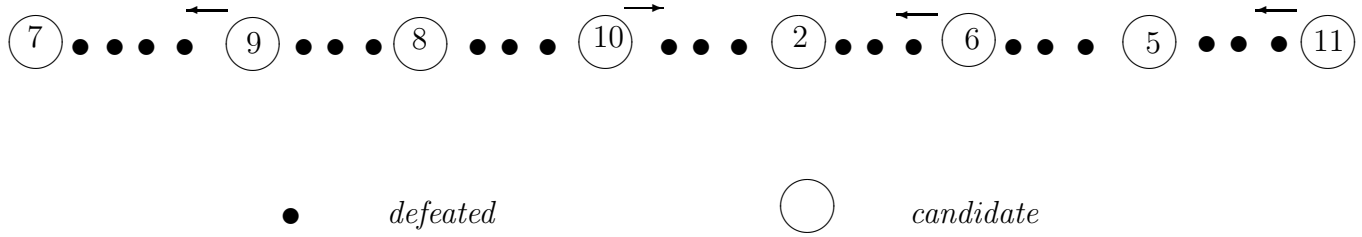


Figure 17: Only some *candidates* will send a Feedback.

Let us integrate the positive feedback idea in the overall strategy of *Stages*: when an "Election" message reaches its target, a positive *feedback* will be sent back to its originator if the id contained in the message is the smallest seen by the target in this stage.

More precisely, when a *candidate* x receives Election messages containing $id(y)$ and $id(z)$ from its neighbouring *candidates*, $y = r(i, x)$ and $z = l(i, x)$, it will send a (positive) *"Feedback"* message: to y if $id(y) < Min\{id(x), id(z)\}$, to z if $id(z) < Min\{id(x), id(y)\}$, to none otherwise. A *candidate* will then survive this stage and enter the new one if and only if it receives a Feedback from both sides.

In the example of Figure 17, *candidates* with ids 2, 5 and 8 will not send any feedback; of these three, only *candidate* with id=2 will enter next stage. The fate of entity with id 7 depends on its other neighbouring *candidate*, which is not shown; so, we do not know whether it will survive or not.

If a node sends a "Feedback" message, it knows that it will not survive this stage. This is the case, in the example, of the entities with ids $6, 9, 10, 11$.

Some entities, however, do not send any "Feedback" and wait for a "Feedback" that will never arrive; this is for example the cases of the entities with ids 5 and 8. How will such an entity discover that no "Feedback" is forthcoming, and it must become *defeated* ? The answer is fortunately simple. Every entity that survives stage i (e.g., the node with id= 2) will start the next stage; its Stage message will act as a *negative feedback* for those entities receiving the message while still waiting in stage i.

More specifically, if while waiting for a "Feedback" message in stage i, an entity receives an "Election" message (clearly with a smaller id) in stage $i + 1$, it becomes *defeated* and forwards the message.

We shall call the protocol *Stages with Feedback*; our description was assuming message ordering. As for protocol *Stages*, this restriction can and will be logically enforced with just local processing.

3.5.1 Correctness

The correctness and termination of the protocol follows from the fact that the entity x_{min} with the smallest identity will always receive a positive feedback from both sides; hence it

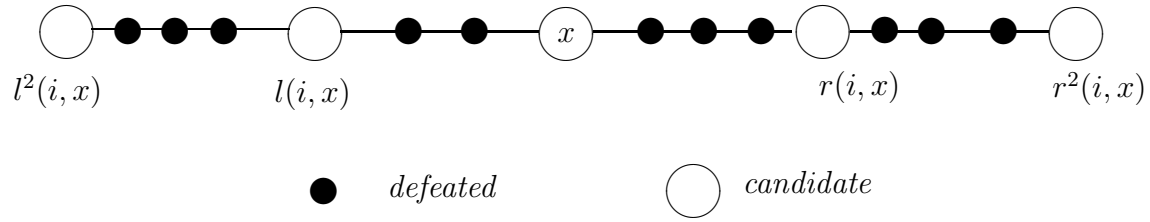


Figure 18: If x survives, those other *candidates* do not.

will never be defeated. At the same time x_{min} never sends a positive feedback; hence its left and right neighbouring candidates in that stage do not survive it. In other words, the number n_i of *candidates* in stage i is monotonically decreasing, and eventually only x_{min} will be in such a state. When this happens, its own "Election" messages will travel along the ring, and termination will be detected.

3.5.2 Messages

We are adding bookkeeping and additional messages to the already highly efficient protocol *Stages*. Let us examine the effect of these changes.

Let us start with the number of stages.

As in *Stages*, if a *candidate* x in stage i survives, it is guaranteed that its neighbouring *candidates* in the same stage, $r(i,x)$ and $l(i,x)$, will become *defeated*. With the introduction of positive feedback, we can actually guarantee that if x survives, also the first *candidate* to the right of $r(i,x)$ will not survive ! and neither will the first candidate to the left of $l(i,x)$.

This is because, if x survives, it must have received a "Feedback" from both $r(i,x)$ and $l(i,x)$. But if $r(i,x)$ sends "Feedback" to x, it does not send one to its neighbouring *candidate* $r^2(i,x)$; similarly, $l(i,x)$ does not send a "Feedback" to $l^2(i,x)$.

In other words,

$$n_i \leq \frac{n_{i-1}}{3}$$

That is, at most one third of the candidates starting a stage will enter the next one. Since $n_1 = n$, the total number of stages σ_{Stages} is at most $\sigma_{Stages} \leq \lceil log_3 n \rceil + 1$. Note that this number is actually achievable: there are initial configurations of the ids that will force the protocol to have exactly these many stages (Exercise 9.22).

In other words, the number of stages has *decreased* with the use of "Feedback" messages. However, we are sending more messages in each stage.

Let us examine now how many messages will be sent in each stage. Consider stage i; this will be started by n_i candidates. Each candidate will send an "Election" message that will travel

to the next *candidate* on either side. Thus, exactly like in *Stages*, two "Election" messages will be sent over each link, one in each direction, for a total of $2n$ "Election" messages per stage. Consider now the "Feedback" messages; a candidate sends at most one "Feedback" and only in one direction. Thus, in the segment of the ring between two candidates, there will be at most one "Feedback" message on each link; hence there will be no more than n "Feedback" transmissions in total in each stage. This means that, in each stage there will be at most $3n$ messages.

Summarizing

$$\mathbf{M}[StageswithFeedback] \leq 3\ n\ log_3 n + O(n) \leq 1.89\ n \log n + O(n) \tag{18}$$

In other words, the use of feedback with the electoral stages allows us to reduce the number of messages in the worst case. the use of *Minimum Initiator* strategy yields the similar result:

$$\mathbf{M}[StageswithFeedback - MinInit] \leq 1.89\ n \log k_* + O(n) \tag{19}$$

In the analysis of the number of "Feedback" messages sent in each stage, we can be more accurate; in fact, there are some areas of the ring (composed of consecutive *defeated* entities between two successive candidates) where no feedback messages will be transmitted at all. In the example of Figure 17, this is the case of the area between the candidates with ids 8 and 10. The number of these areas is exactly equal to the number n_{i+1} of candidates that survive this stage (Exercise 9.19). However, the savings are not enough to reduce the constant in the leading term of the message costs (Exercise 9.21).

Granularity of Analysis: Bit Complexity

The advantage of protocol *Stages with Feedback* becomes more evident when we look at communication costs at a finer level of granularity, focusing on the actual *size* of the messages being used. In fact, while the "Election" messages contain values, the "Feedback" messages are just *signals*, each containing $O(1)$ bits. (recall the discussion in Section 2).

In each stage, only the $2n$ "Election" messages carry a value, while the other n are signals; hence, the total number of *bits* transmitted will be at most

$$2\ n\ (c\ + \log \mathbf{id})\ log_3 n + n\ c\ log_3 n + l.o.t.$$

where **id** denotes the largest value sent in a message, $c = O(1)$ denotes the number of bits required to distinguish among the different types of message, and *l.o.t.* stands for "lower order terms". That is

$$\mathbf{B}[StageswithFeedback] \leq 1.26\ n \log n\ \log \mathbf{id} + l.o.t. \tag{20}$$

The improvement on the bit complexity of *Stages*, where every message carries a value, is thus in the reduction of the constant from 2 to 1.26.

3.5.3 Further improvements ?

The use of electoral stages allows to transform the election process into one of successive "eliminations", reducing the number of *candidates* at each stage. In the original protocol *Stages* each surviving candidate will eliminate its neighbouring candidate on each side, guaranteeing that at least half of the candidates are eliminated in each stage. By using feedback, protocol *Stages with Feedback* extends the "reach" of a candidate also to the second neighbouring candidate on each side, ensuring that at least 2/3 of the candidates are eliminated in each stage. Increasing the "reach" of a candidate during a stage will result in a larger proportion of the candidates in each stage, thus reducing the number of stages. So, intuitively, we would like a candidate to reach as far as possible during a stage. Obviously the price to be paid is the additional messages required to implement the longer reach.

In general, if we can construct a protocol that guarantees a reduction rate of at least b, i.e., $n_i \leq \frac{n_{i-1}}{b}$, then the total number of stages would be $log_b(n)$; if the messages transmitted in each stage are at most an, then the overall complexity will be

$$a\ n\ log_b(n) = \tfrac{a}{\log b}\ n \log n$$

To improve on *Stages with Feedback*, the reduction must be done with a number of messages such that $\frac{a}{\log b} < 1.89$. Whether this is possible, it is an open problem (Problem 9.3).

3.6 Alternating Steps

It should be clear by now that the road to improvement, on which creative ingenuity will travel, is oftentimes paved by a deeper understanding of what is already available.

A way to achieve such an understanding is by examining the functioning of the object of our improvement in "slow motion", so to observe its details.

Let us consider protocol *Stages*. It is rather simple and highly efficient. We have already shown how to achieve improvements by extending the "reach" of a candidate during a stage"; in a sense, this was really "speeding up" the functioning of the protocol. Let us examine now *Stages* instead by "slowing down" its functioning.

In each stage, a candidate sends its id in both direction, receives an id from each direction, and decides whether to survive, be elected, or become defeated based on its own and the received values.

Consider the example shown in Figure 19; the result of *stages* will result in: candidates w, y and v being eliminated; x and z surviving; the fate of u will depend on its right candidate neighbour, which is not shown.

We can obviously think of "sending in both direction" as two separate steps: send to one direction (say "right"), and send to the other. Assume for the moment that the ring is *oriented*: "right" has the same meaning for all entities. Thus, the stage can be though of two steps: (1) the *candidate* sends to the "right" and receives from the "left"; (2) it will then send to the "left" and receive from the "right".

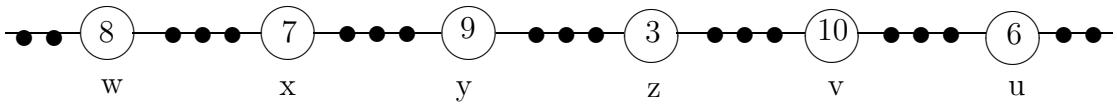


Figure 19: *Alternating Steps*: slowing the execution of *Stages*.

Consider the first step in the same example shown in Figure 19; both *candidates* y and v already know at this time that they do not survive. Let us take advantage of this "early" discovery. We will use each of these two steps to make an electoral decision, and we will eliminate a *candidate* after step (1) if it receives a smaller id in this step. Thus, a *candidate* will perform step (2) only if is not eliminated in step (1).

The advantage of doing so becomes clear observing that, by eliminating in each step of a phase, we eliminate more than in the original phase; in the example of Figure 19, also x will be eliminated !

Summarizing, the idea is that, at each step, a candidate sends only one message with its value, waits for one message, and decides based on its value and the received one; the key is to alternate at each step the direction in which messages are sent.

This protocol, which we shall call *Alternate*, is shown in Fig. 20, where **close** and **open** denote the operation of closing a port (with the effect of enqueueing incoming messages), and opening a closed port (dequeueing the messages), respectively; and the procedures *Initialize* and *Process_Message* are shown in Fig. 21.

Correctness

The correctness of the protocol follows immediately from observing that, as usual, the *candidate* x_{min} with smallest value will never be eliminated and that, on the contrary, will in each step eliminate a neighbouring *candidate*. Hence, the number of *candidates* is monotonically decreasing in the steps; when only x_{min} is left, its message will complete the tour of the ring transforming it into the *leader*. The final notification will ensure proper termination of all entities.

Costs

To determine the cost is slightly more complex. There are exactly n messages transmitted in each step, so we need to determine the total number $\sigma_{Alternate}$ of steps until a single candidate is left, in the worst case, regardless of the placement of the ids in the ring, time delays, etc.

Let n_i be the candidate entities starting step i; clearly $n_1 = n$ and $n_k = 1$. We know that two successive steps of *Alternate* will eliminate more candidates than a single stage of *Stages*; hence, the total number of steps $\sigma_{Alternate}$ (or, where no confision arises, simply σ) will be less than twice the number of stages of *Stages*:

$$\sigma_{Alternate} < 2 \log n.$$

We can however be more accurate on the amount of elimination is performed in two successive steps.

PROTOCOL Alternate.

- States: $\mathcal{S}$ = {ASLEEP, CANDIDATE, DEFEATED, FOLLOWER, LEADER};
  $\mathcal{S}_{INIT}$ = {ASLEEP};
  $\mathcal{S}_{TERM}$ = {FOLLOWER, LEADER}.
- Restrictions: **RI** $\cup$*OrientedRing* $\cup$ $MessageOrdering$.

```
ASLEEP
      Spontaneously
      begin
           INITIALIZE;
           become CANDIDATE;
      end

      Receiving(``Election'', id*, step*)
      begin
           INITIALIZE;
           become CANDIDATE;
           PROCESS_MESSAGE;
      end


CANDIDATE
        Receiving(``Election'', id*, step*)
        begin
             if id* ≠ id(x) then
               PROCESS_MESSAGE;
             else
               send(Notify) to N(x);
               become LEADER;
        end


DEFEATED
        Receiving(⋆)
        begin
             send(⋆) to other;
             if ⋆ = Notify then become FOLLOWER endif;
        end
```

Figure 20: Protocol *Alternate*

```
Procedure INITIALIZE
begin
     step:= 1;
     min:= id(x);
     send(``Election'', id(x), step) to right;
     close(right);
end

Procedure PROCESS_MESSAGE
begin
     if id*< min then
       open(other);
       become DEFEATED;
     else
       step:= step+1;
       send(``Election'', id(x), step) to sender;
       close(sender);
       open(other);
     endif
end
```

Figure 21: Procedures used by protocol *Alternate*

Assume that in step i the direction is "right" (thus, it will be "left" in step $i+1$). Let d_i denote the number of candidates that are eliminated in step i. Of those n_i candidates which start step i, d_i will be defeated and only n_{i+1} will survive that step. That is,

$$n_i = d_i + n_{i+1}$$

Consider a candidate x that survives both step i and step $i+1$. First of all observe that the candidate to the right of x in step i will be eliminated in that step. (If not, it would mean that its id is smaller than $id(x)$ and thus would eliminate x in step $i+1$; but we know that x survives.)

This means that every candidate which, like x, survives both stages, will eliminate one candidate in the first stage; in other words,

$$d_i \geq n_{i+2}$$

but then

$$n_i \geq n_{i+1} + n_{i+2} \tag{21}$$

The consequence of this fact is very interesting. In fact, we know that $n_\sigma = 1$ and, obviously, $n_{\sigma-1} \geq 2$. From Equation 21 we have $n_{\sigma-i} \geq n_{\sigma-i+1} + n_{\sigma-i+2}$.

Consider now the Fibonacci numbers F_j defined by $F_j = F_{j+1} + F_{j+2}$, where $F_{-1} = 0$ and $F_0 = 1$. Then, clearly

$$n_{\sigma-i} \geq F_{i+1}$$

It follows that $n_1 \geq F_\sigma$, but $n_1 = n$; thus σ is the index of the largest Fibonacci number not exceeding n. Our goal is to determine σ, the number of steps until there is only one candidate left; now we can. Since $F_j = b\ (\frac{1+\sqrt{5}}{2})^j$ where b is a positive constant, we have

$$n \geq F_\sigma = b\ (\tfrac{1+\sqrt{5}}{2})^\sigma$$

from where

$$\sigma_{Alternate} \leq 1.44 \log n + O(1)$$

That means that after *at most* so many steps, there will be only one candidate left. Observe that this we have derived is actually *achievable*. In fact, there are allocations of the ids to the nodes or a ring which will force the protocol to perform $\sigma_{Alternate}$ steps before there is only one *candidate* left (Exercise 9.26). In the next step, this *candidate* will become *leader* and start the notification. These last two operations require n messages each,

Thus the total number of messages will be

$$\mathbf{M}[Alternate] \leq 1.44\ n \log n + O(n) \tag{22}$$

In other words, protocol *Alternate* is not only simple but also more efficient that all other protocols seen so far.

Recall however that it has been described and analyzed assuming that the ring is *oriented*.

Question. *What happens if the ring is* not *oriented ?*

If the entities have different meaning for "right", when implementing the first step, some candidates will send messages clockwise while others in a counterclockwise direction.

Notice that, in the implementation for oriented rings described above, this would lead to deadlock, because we close the port we are not waiting to receive from; the implementation can be modified so that ports are never closed (Exercise 9.24). Consider this to be the case.

It will then happen that a candidate waiting to receive from "left" will instead receive from "right". Call this situation a *conflict*.

What we need to do is to add to the protocol a *conflict resolution* mechanism to cope with such situations. Clearly this complicates the protocol and increases the number of messages (Problem 9.2).

3.7 Unidirectional Protocols

The first two protocols we have examined, *All the Way* and *AsFar*, did not really require the restriction Bidirectional Links; in fact, they can be used without any modification in

a *directed* or *unidirectional* ring. The subsequent protocols, *Distances*, *Stages*, *Stages with Feedback*, and *Alternate* all used the communication links in both directions; e.g., for obtaining feedback. It was through them that we have been able to reduce the costs from $O(n^2)$ to a guaranteed $O(n \log n)$ messages. The immediate and natural question is:

Question: *Is "Bidirectional Links" necessary for a* $O(n \log n)$ *cost ?*

The question is practically relevant because, if the answer is positive, it would indicate that an additional investment in communication hardware (i.e., full duplex lines) is necessary to reduce the operating costs of the election task. The answer is important also from a theoretical point of view because, if positive, it would clearly indicate the "power" of the restriction Bidirectional Links. Not surprisingly, this question has attracted the attention of many researchers.

We are going to see now that the answer is actually: *No* !

We are also going to see that, strangely enough, we know how to do better with unidirectional links than with bidirectional ones !

First of all, we are going to show how the execution of protocols *Stages* and *Alternate* can be *simulated* in a unidirectional links yielding the same (if not better) complexity. Then, using the lessons learned in this process, we are going to develop a more efficient *unidirectional* solution.

3.7.1 Unidirectional Stages

What we are going to do is to show how to *simulate* the execution of protocol *Stages* in unidirectional rings $\vec{R}$, with the same message costs.

Consider how protocol *Stages* work. In a stage, a *candidate* entity x
(1) sends a message carrying a value $v(x)$ (its id) in both directions, and thus receives a message with the value (the id) of another *candidate* from each directions; then,
(2) based these three values (i.e., its own and the two received ones), makes a decision on whether it (and its value) should survive this stage and start the next stage.

Let us implement each of these two steps separately.

Step (1) is clearly the difficult one since, in a unidirectional ring, messages can only be sent in one direction. Decompose the operation "send in both direction" into two substep: (I) "send in one direction", and then (II) "send in the other direction".
Now, substep (I) can be executed directly in $\vec{R}$; as a result, every *candidate* will receive a message with the value of its neighbouring *candidate* from the opposite direction (see Figure 22 *c*)). The problem is in implementing now substep (II); since we cannot send in the other direction, *we will send again in the same direction*, and, since it is meaningless to send again the same information, *we will send the information we just received*. As a result, every *candidate* will receive now the value of another candidate from the opposite direction (see Figure 22 *a*)).

Every entity in $\vec{R}$ has now three values at its disposal: the one it started with. plus the two

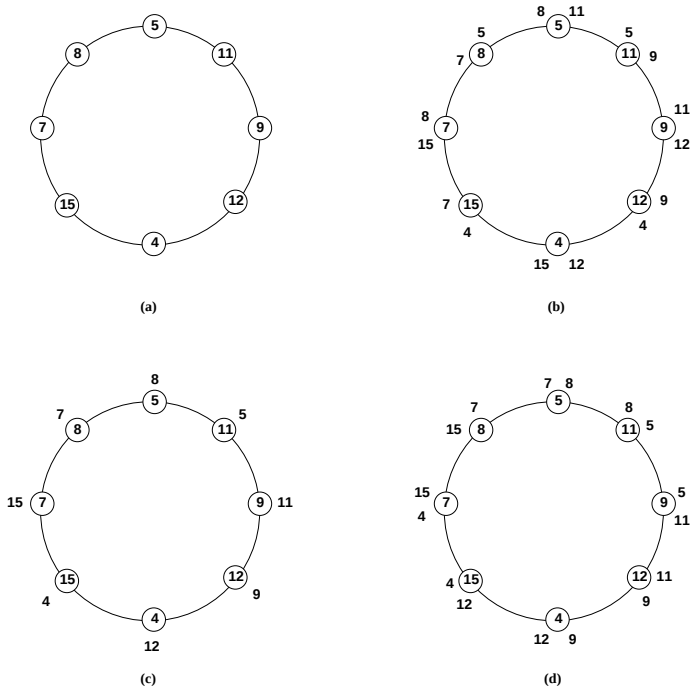


Figure 22: (a) Initial configuration. (b) Information after the first full stage of *Stages* with bidirectional links. (c) Information after first substep in the unidirectional simulation. (d) Information after the second substep

received ones. We can now proceed to implement Step (1). To simulate the bidirectional execution, we need that a *candidate* decides on whether to survive or to became *passive* based on exactly *the same information* in $\vec{R}$ as in the bidirectional case. Consider the initial configuration in the example shown in Figure 22 and focus on the *candidate* x with starting value 7; in the bidirectional case, x decides that the value 7 should survive based on the information: 7, 15 and 8. In the unidirectional case, after the implementation of Step (1), x knows now 4 and 15 in addition to 7. This is *not the same information* at all ! In fact, it would lead to totally different decisions in the two cases, destroying the simulation.

There is however in $\vec{R}$ a *candidate* which, at the end of Stage (1), has exactly the same information that x has at the end of Stage (1) in the bidirectional case: this is the candidate that started with value 8. In fact, the information available in R exists in $\vec{R}$ (compare carefully Figures 22 (b) and (d)), but it is *shifted to the "next"* candidate in the ring direction ! It is thus possible to make the same decisions in $\vec{R}$ as in R; they will just have be made by different entities in the two cases.

In each stage, a *candidate* makes a decision on a value. In protocol *Stages*, this value was always the *candidate*'s id. In the unidirectional algorithm this value is not the id; it is *the first value sent by its neighbouring candidate* in Step (1). We will call this value the *envelope*.

IMPORTANT. Be aware that, unless we add the assumption Message Ordering, it is possible that the second value arrives before the envelope. This problem can be easily solved; e.g., by having a bit in the message indicating its order (first or second) of transmission within the stage.

It is not difficult to verify that the simulation is *exact*: in each stage, exactly the same values survive in $\vec{R}$ as in R; thus, the number of stages is exactly the same. The cost of each stage is also the same: $2n$ messages. In fact, each node will send (or forward) exactly two messages.

In other words,

$$\mathbf{M}[UniStages] \leq 2\ n \log n + O(n) \tag{23}$$

This shows that $O(n \log n)$ guaranteed message costs can be achieved in ring networks also *without* Bidirectional Links.

The corresponding protocol *UniStages* is shown in Fig. 23, described not as a unidirectional simulation of *Stages* (which indeed it is) but directly as a unidirectional protocol.

NOTES. In this implementation:
(1) we elect a leader only among the initiators (using approach *minimum initiator*);
(2) message ordering is not assumed; within a stage, we use a Boolean variable, *order* to distinguish between value and envelope; to cope with messages from different stages arriving out of order: if a *candidate* receives a message from the "future" (i.e., with a higher stage number), it will be transformed immediately into *defeated* and forward the message.

3.7.2 Unidirectional Alternate

We have shown how to simulate *Stages* in a unidirectional ring, achieving exactly the same cost. Let us focus now on *Alternate*; this protocol makes full explicit use of the full duplex communication capabilities of the bidirectional ring by alternating direction at each step. Surprisingly, it is possible to achieve an exact simulation also of this protocol in a unidirectional ring $\vec{R}$.

Consider how protocol *Alternate* works. In a "left" step, a *candidate* entity x
(1) sends a message carrying a value $v(x)$ to the "left", and receives a message with the value of another *candidate* from the "right"
(2) based on these two values (i.e., its own and the received one), x makes a decision on whether it (and its value) should survive this step and start the next step.

The actions in a "right" step are the same except that "left" and "right" are interchanged.

Consider the ring $\vec{R}$ shown in Figure 25, and assume we can send messages only to "right". This means that the initial "right" step can be trivially implemented: every entity will send a value (its own) and receive another; it starts the next step if and only if the value it receives is not smaller that its own.

Let us concentrate on the "left" step. Since a candidate cannot send to the left, it will have

PROTOCOL UniStages.

- States: $\mathcal{S}$ = {ASLEEP, CANDIDATE, DEFEATED, FOLLOWER, LEADER};
  $\mathcal{S}_{INIT}$ = {ASLEEP};
  $\mathcal{S}_{TERM}$ = {FOLLOWER, LEADER}.
- Restrictions: $\{Connectivity, TotalReliability, UnidirectionalRing\}$.

```
ASLEEP
      Spontaneously
      begin
           INITIALIZE;
           become CANDIDATE;
      end

      Receiving(‘‘Election’’, value*, stage*,order*)
      begin
           send (‘‘Election’’, value*, stage*, order*);
           become DEFEATED;
      end


CANDIDATE
      Receiving(‘‘Election’’, value*, stage*, order*)
      begin
           if   value* ≠ value1 then
                PROCESS_MESSAGE;
           else
                send(Notify);
                become LEADER;
      end

DEFEATED
      Receiving(⋆)
      begin
           send(⋆);
           if ⋆ = Notify then become FOLLOWER endif;
      end
```

Figure 23: Protocol *Unistages*

```
Procedure INITIALIZE
begin
     stage:= 1;
     count:= 0;
     order:= 0;
     value1:= id(x);
     send(‘‘Election’’, value1, stage, order);
end

Procedure PROCESS_MESSAGE
begin
     if  stage* = stage then
         if  order* = 0 then
             envelope:= value*;
             order:= 1;
             send (‘‘Election’’, value*, stage*, order);
         else
             value2:= value*;
         endif
         count:=count+1;
         if count=2 then
             if  envelope < Min(value1, value2) then
                 order:= 0;
                 count:= 0;
                 stage:= stage+1;
                 value1:= envelope;
                 send (‘‘Election’’, value1, stage, order);
             else
                 become DEFEATED;
             endif
         endif
     else
         if stage* > stage then
             send (‘‘Election’’, value*, stage*, order*);
             become DEFEATED;
         endif
     endif
end
```

Figure 24: Procedures used by protocol *Unistages*

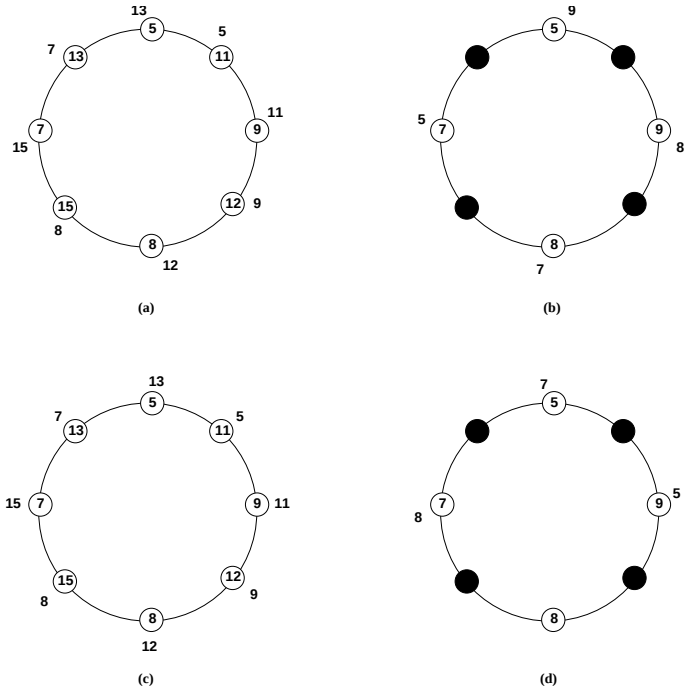


Figure 25: (a-b) Information after (a) the first step and (b) the second step of *Alternate* in an oriented bidirectional ring. (c-d) Information after (c) the first step and (d) the second step of the unidirectional simulation.

to send to the "right'. Let us do so. Every candidate in $\vec{R}$ has now two values at its disposal: the one it started with, plus the received one.

To simulate the bidirectional execution, we need that a candidate makes a decision on whether to survive or to became passive based on exactly *the same information* in $\vec{R}$ as in the bidirectional case. Consider the initial configuration in the example shown in Figure 25. First of all observe that the information in the "right" step is the same both in the bidirectional (a) and in the unidirectional (c) case. The differences occur in the "left" step.

Focus on the candidate x with starting value 7; in the second step of the bidirectional case, x decides that the value 7 should not survive based on the information: 5 and 7. In the unidirectional case, after the second step, x knows now 7 and 8. This is *not the same information* at all ! In fact, it would lead to totally different decisions in the two cases, destroying the simulation.

There is however in $\vec{R}$ a candidate which, at the end of the second step, has exactly the same information that x has in the bidirectional case: this is the candidate that started with value 5. As we have seen already in the simulation of *Stages*, the information available in R exists in $\vec{R}$ (compare carefully Figures 25(b) and (d). It is thus possible to make the same decisions in $\vec{R}$ as in R; they will just have be made by different entities in the two cases.

Summarizing, in each step, a *candidate* makes a decision on a value. In protocol *Alternate*, this value was always the candidate's id. In the unidirectional algorithm this value changes depending on the step. Initially, it is its own value; in the "left" step it is the value it receives; in the "right" step it is the value it already has.

In other words:

1. in the "right" step, a *candidate* x survives if and only the received value is *larger* than $v(x)$;
2. in the "left" step, a *candidate* x survives if and only if the received value is *smaller* than $v(x)$, and if so, x will now play for that value.

Working out a complete example will help clarify the simulation process and dispel any confusion (Exercise 9.33).

IMPORTANT. Be aware that, unless we add the assumption Message Ordering, it is possible that the value from step $i+1$ arrives before the value for step i. This problem can be easily solved; e.g., by having a bit in the message indicating its order (first or second) of transmission within the stage.

It is not difficult to verify that the simulation is *exact*: in each step, exactly the same values survive in $\vec{R}$ as in R; thus, the number of steps is exactly the same. The cost of each step is also the same: n messages. Thus,

$$\mathbf{M}[UniAlternate] \leq 1.44\ n \log n + O(n) \tag{24}$$

The unidirectional simulation of *Alternate* is shown in Fig. 26; it has been simplified so that we elect a leader only among the initiators, and assuming *Message Ordering*. The protocol can be modified to remove this assumption without changes in its cost (Exercise 9.34). The procedures *Initialize* and *Prepare_Message* are shown in Fig. 27.

3.7.3 An Alternative Approach

In all the solutions we have seen so far, both for unidirectional and bidirectional rings, we have use the same basic strategy of *minimum finding*; in fact in all or protocols so far, we have electing as a leader the entity with smallest value (either among all the entities or just the initiators). Obviously, we could have used *maximum finding* in those solution protocols, just substituting the function *Min* with *Max*, and obtaining the exact same performance.

A very different approach consists in mixing these two strategies. More precisely, consider the protocols based on *electoral stages*. In all of them, what we could do is to alternate strategy in each stage: in "odd" stages we use the function *Min*, and in "even" stages we use the function *Max*. Call this approach *min-max*.

It is not difficult to verify that all the stage-based protocols we have seen so far, both bidirectional and unidirectional, still correctly solve the election problem; moreover, they do so with the same costs as before (Exercises 9.11, 9.23, 9.28, 9.31, 9.36).

PROTOCOL UniAlternate.

- States: $\mathcal{S}$ = {ASLEEP, CANDIDATE, DEFEATED, FOLLOWER, LEADER};
  $\mathcal{S}_{INIT}$ = {ASLEEP};
  $\mathcal{S}_{TERM}$ = {FOLLOWER, LEADER}.
- Restrictions: $\{Connectivity, TotalReliability, UnidirectionalRing, MessageOrdering\}$.

```
ASLEEP
      Spontaneously
      begin
           INITIALIZE;
           become CANDIDATE;
      end

      Receiving(``Election'', value*, stage*,order*)
      begin
           send (``Election'', value*, stage*, order*);
           become DEFEATED;
      end


CANDIDATE
      Receiving(``Election'', value*, stage*)
      begin
           if   value* ≠ value then
                PROCESS_MESSAGE;
           else
                send(Notify);
                become LEADER;
      end

DEFEATED
      Receiving(⋆)
      begin
           send(⋆);
           if ⋆ = Notify then become FOLLOWER endif;
      end
```

Figure 26: Protocol *UniAlternate*

```
Procedure INITIALIZE
begin
     step:= 1;
     direction:= "right";
     value:= id(x);
     send(‘‘Election’’, value, step, direction);
end

Procedure PROCESS_MESSAGE
begin
     if  direction = "right" then
         if  value < value* then
             step:= step+1;
             direction:= "left";
             send (‘‘Election’’, value, step, direction);
         else
             become DEFEATED;
         endif
     else
         if  value > value* then
             step:= step+1;
             direction:= "right";
             send (‘‘Election’’, value, step, direction);
         else
             become DEFEATED;
         endif
     endif
end
```

Figure 27: Procedures used by protocol *UniAlternate*

The interesting and surprising thing is that this approach can lead to the design of a more efficient protocol for unidirectional rings.

The protocol we will construct has a simple structure. Let us assume that every entity starts and that there is *Message Ordering* (we will remove both assumptions later).

1. Each initiator x becomes *candidate*, prepares a message containing its own value $id(x)$ and the stage number $i = 1$, and sends it (recall, we are in a unidirectional ring, so there is only one out-neighbour); x is called the *originator* of this message and remembers its content.

2. When a message with value b arrives at a *candidate* y, y compares the received value b to the value a it sent in its last message.

   (a) If $a = b$, the message originated by y has made a full trip around the ring; y becomes the *leader* and notifies all other entities of termination.

   (b) If $a \neq b$,the action y will take depends on the stage number j:

      i. if j is "even", the message is *discarded* if and and only if $a < b$ (i.e., b survives only if *max*);

      ii. if j is "odd", the message is *discarded* if and and only if $a > b$ (i.e., b survives only if *min*).

   If the message is discarded, y becomes *defeated*; otherwise, y will enter the next stage: originate a message with content $(b, j + 1)$ and send it.

3. A *defeated* entity will, as usual, forward received messages.

For an example, see Fig. 28.

The correctness of the protocol follows from observing that,
(a) in an even stage i, the *candidate* x receiving the largest of all values in that stage, $v_{max}(i)$, will survive and enter the next stage; on the other hand, its "predecessor" $l(i, x)$ that originated that message will become *defeated* (Exercise 9.37). and
(b) in an odd stage j, the *candidate* y receiving the smallest of all values in that stage, $v_{min}(j)$, will survive and and enter the next stage; furthermore, its "predecessor" $l(j, y)$ that originated that message will become *defeated*.
In other words, in each stage at least one *candidate* will survive that stage, and the number of *candidates* in a stage is monotonically decreasing with the number of stages. Thus, within finite time, there will only one *candidate* left; when that happens, its message returns to it transforming it into *leader*.

IMPORTANT Note that the entity that will be elected *leader* will not be the one with the smallest value nor the one with the largest value.

Let us now consider the costs of this protocol, we will call *MinMax*. In a stage, each *candidate* sends a message that travels to the next *candidate*. In other words, in each stage there will be exactly n messages. Thus, to determine the total number of messages, we need to compute the number σ_{MinMax} of stages.

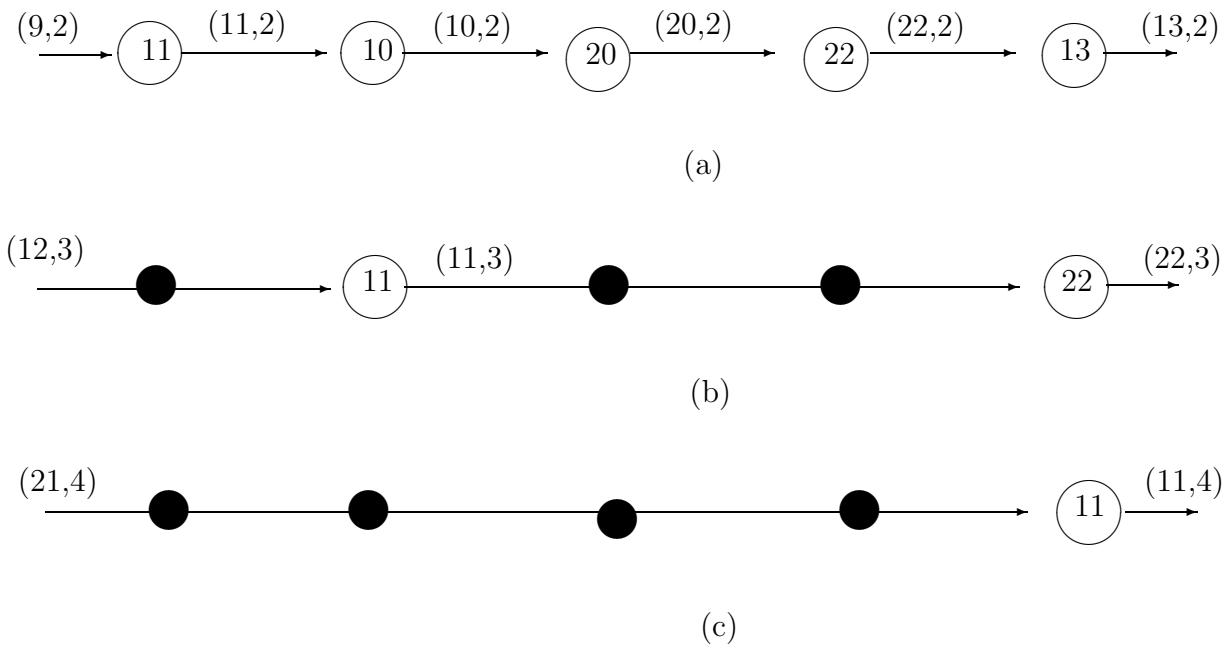


Figure 28: Protocol *MinMax*: (a) In an even stage, a *candidate* survives only if it receives an envelope with a larger value; (b) it then generates an envelope with that value and starts the next stage. (c) In an odd stage, a candidate survives only if it receives an envelope with a smaller value; if so, it generates an envelope with that value and starts the next stage.

We can rephrase the protocol in terms of values instead of entities. Each value sent in a stage j travels from its originator to the next *candidate* in stage j. Of all these values, only some will survive and be sent in the next stage: in an even stage, a value survives if it is *larger* than its "successor" (i.e., the next value in the ring also this stage); similarly, in an odd stage, it survives if it is *smaller* than its successor. Let n_i be the number of *values* in stage i; of those, d_i will be *discarded* and n_{i+1} will be sent in the next stage. That is,

$$n_{i+1} = n_i - d_i$$

Let i be an odd (i.e., *min*) stage, and let value v survive this stage; this means that the successor of v in stage i, say u, is larger than v; i.e., $u > v$. Let v survive also stage $i+1$ (an even, i.e. *max*, stage). This implies that that v must have been discarded in stage i: if not, the entity that originates the message $(i+1, u)$ would discard $(i+1, v)$ because $u > v$; but we know that x survives this stage. This means that every value that, like v, survives both stages, will eliminate one value in the first of the two stages; in other words,

$$n_{i+2} \leq d_i$$

but then

$$n_i \geq n_{i+1} + n_{i+2} \tag{25}$$

Notice that this is exactly the same equation as the one (Equation 21) we derived for protocol *Alternate*. We thus obtain that

$$\sigma_{MinMax} \leq 1.44 \log n + O(1)$$

After at most these many stages, there will be only one value left. Observe that this bound we have derived is actually *achievable*. In fact, there are allocations of the ids to the nodes or a ring which will force the protocol to perform σ_{MinMax} steps before there is only one value left (Exercise 9.38). The *candidate* sending this value will receive its message back and become *leader*; it will then start the notification. These last two steps require n messages each; thus the total number of messages will be

$$\mathbf{M}[MinMax] \leq 1.44\ n \log n + O(n) \tag{26}$$

In other words, we have been able to obtain the same costs of *UniAlternate* with a very different protocol, *MaxMin*, described in Fig. 29.

We have assumed that all entities start. When removing this assumption we have two options: the entities that are not initiators can be 1. made to start (as if they were initiators) upon receiving their first message; or 2. transformed into *passive* and just act as relayers. The second option is the one used in Fig. 29.

We have also assumed message ordering in our discussion. As with all the other protocols we have considered, this restriction can be enforced with just local bookkeeping at each entity, without any increase in complexity (Exercise 9.39).

Hacking: Employing the Defeated

The different approach used in protocol *MinMax* has led to a different way of obtaining the same efficiency as we had already with *UniAlternate*. The advantage of *MinMax* is that it is possible to obtain additional improvements that lead to a significantly better performance.

Observe that, like in most previous protocols, the *defeated* entities play a purely passive role; i.e., they just forward messages. The key observation we will use to obtain an improvement in performance is that these entities can be exploited in the computation.

Let us concentrate on the even stages, and see if we can obtain some savings for those steps. The message sent by a *candidate* travels (forwarded by the *defeated* entities) until it encounters the next *candidate*. This distance can vary, and be very large. We we will do is to control the maximum distance to which the transfer will travel, following the idea we developed in section 3.3.

(I) In an even step j, a message will travel no more than a predefined distance $dis(j)$.

This is implemented by having in the message a counter (initially set to $dis(j)$) that will be decremented by one by each defeated node it passes. What is the appropriate choice of $dis(i)$ will be discussed next.

PROTOCOL MinMax.

- States: $\mathcal{S}$ = {ASLEEP, CANDIDATE, DEFEATED, FOLLOWER, LEADER}; $\mathcal{S}_{INIT}$ = {ASLEEP}; $\mathcal{S}_{TERM}$ = {FOLLOWER, LEADER}.
- Restrictions: $\{Connectivity, TotalReliability, MessageOrdering\}$.

```
ASLEEP
      Spontaneously
      begin
           stage:= 1; value:= id(x);
           send(‘‘Envelope’’, value, stage);
           become ORIGINATOR;
      end

      Receiving(‘‘Envelope’’, value*, stage*)
      begin
           send (‘‘Envelope’’, value*, stage*);
           become DEFEATED;
      end


CANDIDATE
      Receiving(‘‘Envelope’’, value*, stage*)
      begin
           if   value* ≠ value then
                PROCESS_ENVELOPE;
           else
                send(Notify);
                become LEADER;
      end



DEFEATED
        Receiving(‘‘Envelope’’, value*, stage*)
        begin
             send(‘‘Envelope’’, value*, stage*);
        end


        Receiving(‘‘Notify’’)
        begin
             send (‘‘Notify’’);
             become FOLLOWER;
        end
```

Figure 29: Protocol *MinMax*

```
Procedure PROCESS_ENVELOPE
begin
     if  odd(stage*) then
         if  value* < value then
             stage= stage+1;
             value:= value*;
             send (‘‘Envelope’’, value*, stage);
         else
             become DEFEATED;
     else
         if  value* > value then
             stage= stage+1;
             value:= value*;
             send (‘‘Envelope’’, value, stage);
         else
             become DEFEATED;
         endif
     endif
end
```

Figure 30: Procedure Process_Envelope of Protocol *MinMax*

Every change we make in the protocol has strong consequences. As consequence of (I), the message from x might not reach the next *candidate* y if it is too far away (more than $dis(j)$) (see Figure 31). In this case, the *candidate* y does *not* receive the message in this stage and thus does not know what to do for the next stage.

IMPORTANT. It is possible that *every* candidate is too far away from the next one in this stage, and hence *none* of them will receive a message.

However, if *candidate* y does *not* receive the message from x, it is because the counter of the message containing (v, j) reaches 0 at a *defeated* node z, on the way from x to y (see Figure 31). To ensure *progress* (i.e., absence of deadlock), we will make that *defeated* z become *candidate* and start the next stage $j + 1$ immediately, sending $(v, j + 1)$. That is,

(II) In an even step j, if the counter of the message reaches 0 at a *defeated* node z, then z becomes *candidate* and start stage $j + 1$ with $value = v*$, where $v*$ is the value in the transfer message.

In other words, we are bringing some *defeated* nodes back into the game making them *candidate* again. This operation could be dangerous for the complexity of the protocol, as the number of *candidates* appears to be increasing (and not decreasing) ! This is easily taken care of: those *originators*, like y, waiting for a transfer message that will not arrive, will become *defeated*.

Question. How will y know that is *defeated* ?

The answer is simple. The *candidate* that starts the next stage (e.g., z in our example) sends a message; when this message reaches a *candidate* (e.g., y) still waiting for a message from the previous stage, that entity will understand, become *defeated*, and forward the message.

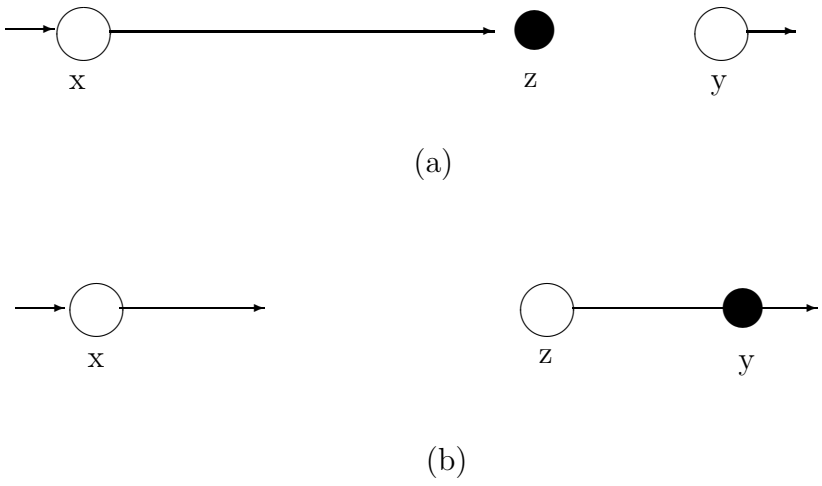


Figure 31: Protocol *MinMax+*. Controlling the distance: in even stage j, the message does not travel more than $dis(j)$ nodes. (a) If it does not reach the next *candidate* y, the *defeated* node reached last, z, will become *candidate* and start the next step; (b) in the next step, the message from z transforms into *defeated* the entity y still waiting for the stage j message.

In other words,

(III) When, in an even step, an *candidate* receives a message for the next step, it becomes *defeated* and forwards the message.

We are giving decisional power to the *defeated* nodes, even bringing some of them back to "life". Let us push this concept forward. and see if we can obtain some other savings.

Let us concentrate on the odd stages.

Consider an even stage i in *MinMax* (eg., Figure 28(a)). Every *candidate* x sends its message containing the value and the stage number, and receives a message; it becomes *defeated* if the received value is smaller than the one it sent. If it survives, x starts stage $i+1$: it sends a message with the received value and the new stage number (see Figure 28(b)); this message will reach the next *candidate*.

Concentrate on the message $(11, 3)$ in Figure 28(b) sent by x. Once $(11, 3)$ reaches its destination y, since $11 < 22$ and we are in a odd (i.e., *min*) stage, a new message $(11, 4)$ will be originated. Observe that the fact that $(11, 4)$ must be originated can be discovered *before* the message reaches y; see Fig. 32. In fact, on its travel from x to y, message $(11, 3)$ will reach the *defeated* node z, that originated $(20, 2)$ in the previous stage; once this happens, z knows that 11 will survive this stage. (Exercise 9.40). What z will do is to become *candidate* again and immediately send $(11, 4)$.

(IV) When, in an even stage, a *candidate* becomes *defeated*, it will remember the stage number and the value it sent. If, in the next stage, it receives a message with a smaller value, it will become *candidate* again and start the next stage with that value.

In our example, this means that the message $(11, 3)$ from x will stop at z and never reach y;

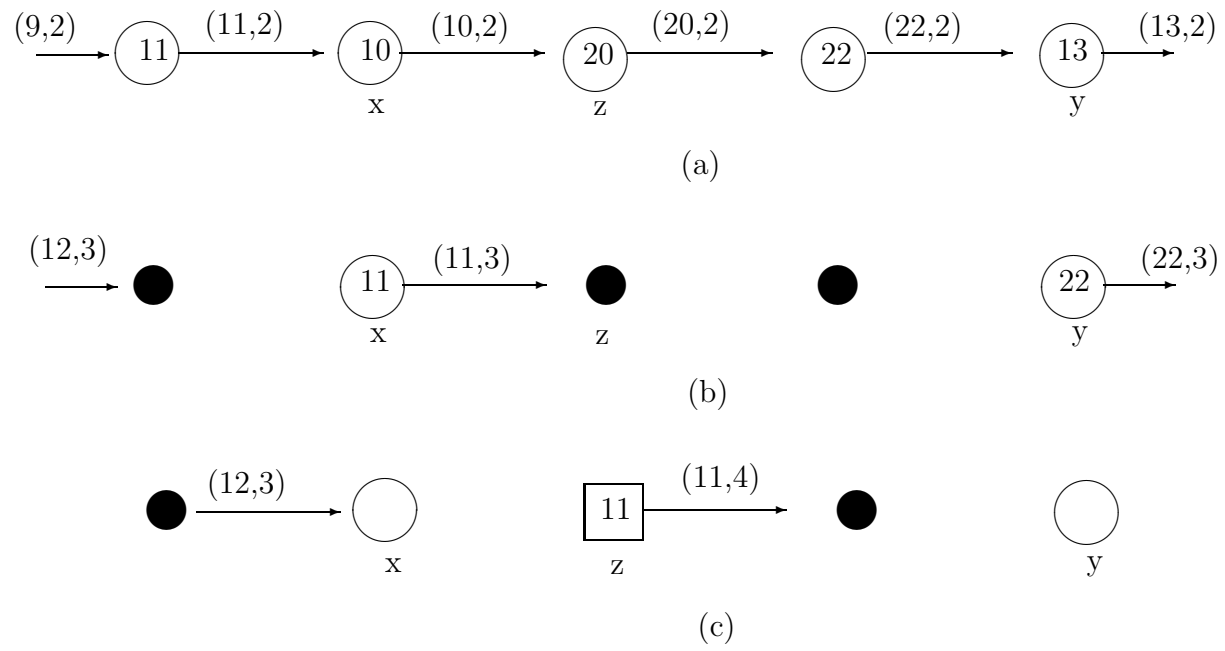


Figure 32: Protocol *MinMax+*. Early promotion in odd stages. (b) The message $(11, 3)$ from x, on its way to y reaches the *defeated* node z that originated $(20, 2)$. (c) Node z becomes *candidate* and immediately originates envelope $(11, 4)$.

thus, we will save $d(z, y)$ messages. Notice that in this stage *every message with a smaller value will be stopped earlier*. We have however transformed a *defeated* entity into a *candidate*. This operation could be dangerous for the complexity of the protocol, as the number of *candidates* appears to be increasing (and not decreasing) ! This is easily taken care of: those *candidates*, like y, waiting for a message of an odd stage that will not arrive, will become *defeated*.

How will y know that is *defeated* ? The answer again is simple. The *candidate* that starts the next stage (e.g., z in our example) sends the message; when this message reaches an entity still waiting for a message from the previous stage (e.g., y), that entity will understand, become *defeated*, and forward the message. In other words,

(V) When, in an odd step, a *candidate* receives a message for the next step, it becomes *defeated* and forwards the message.

The modifications to *MinMax* described by (I)-(V) generate a new protocol that we shall call *MinMax+*. (Exercises 9.41 and 9.42.)

Messages

Let us estimate the cost of protocol *MinMax+*. First of all observe that, in protocol *MinMax*, in each stage a message (v, i) would always reach the next *candidate* in that stage. This is not necessarily so in *MinMax+*. In fact, in an even stage i no message will travel more than $dis(i)$, and in an odd stage a message can be "promoted" by a *defeated* node on the way. We must concentrate on the savings in each type of stages.

Consider a message (v, i); denote by $h_i(v)$ the *candidate* that originates it, and if the message is discarded in this stage, denote by $g_i(v)$ the node that discards it.

For the even stages, we must first of all choose the maximum distance $dis(i)$ a message will travel. We will use

$$dis(i) = F_{i+2}$$

With this choice of distance, we have two very interesting properties.

Property 3.1 *Let i be even.*
If message (v, i) is discarded in this stage, then $d(h_i(v), g_i(v)) \geq F_i$.
For any message $(v, i+1)$, $d(h_i(v), h_{i+1}(v)) \geq F_{i+1}$.

This property allows us to determine the number of stages $\sigma_{MinMax+}$: in a even stage i the distance travelled by any message in stage i is at least F_i; however, none of this messages travels beyond the next *candidate* in the ring. Hence, the distance between two successive *candidates* in odd stage i is at least F_i; this means that the number n_i of *candidates* is at most $n_i \leq \frac{n}{F_i}$. Hence, the number of stages will be at most $F_n^{-1} + O(1)$, where F_n^{-1} is the smallest integer j such that $F_j \geq n$. Thus the algorithm will use at most

$$\sigma_{MinMax+} \leq 1.44 \log n + O(1)$$

stages. This is the same as protocol *MinMax* !

The property also allows us to measure the number of messages we save in the odd stages. In our example of Fig. 32(b), message $(11, 3)$ from x will stop at z and never reach y; thus, we will save $d(z, y)$ transmissions. In general, a message with value v that reaches an even stage $i+1$ (e.g., $(11, 4)$) saves at least F_i transmissions in stage i (Exercise 9.44). The total number of transmissions in an odd stage i is thus at most

$$n - n_{i+1} F_i$$

where n_{i+1} denotes the number of *candidates* in stage $i+1$.

The total number of messages in an even stage is at most n. Since in an even stage $i+1$ each message travels at most F_{i+3} (by Property 3.1, the total number of message transmissions in an even stage $i+1$ will be at most $n_{i+1} F_{i+3}$. Thus, the total number of messages in even $i+1$ is at most

$$Min\{n, n_{i+1} F_{i+3}\}.$$

If we now consider an odd stage i followed by an even stage $i+1$, the total number of message transmissions in the two stages will be at most

$$Min\{n + n_{i+1}(F_{i+3} - F_i), 2n - n_{i+1} F_i\} \leq 2n - n\frac{F_i}{F_{i+3}} < n(4 - \sqrt{5} + \phi^{-2i}).$$

where $\phi = \frac{1+\sqrt{5}}{2}$. Hence,

$$\mathbf{M}[MinMax+] \leq \frac{4-\sqrt{5}}{2} n log_{\phi}(n) + O(n) < 1.271\ n \log n + O(n) \tag{27}$$

Thus, protocol *MinMax+* is the most efficient protocol we have seen so far, with respect to the worst case.

3.8 ⋆ Limits to Improvements

Throughout the previous sections, we have reduced the message costs further and further using new tools or combining existing ones. A natural question is *how far can we go* ? Considering that the improvements have only been in the multiplicative constant of the $n \log n$ factor, the next question becomes: is there a tool or technique which would allow to reduce the message costs for election significantly, e.g. from $O(n \log n)$ to $O(n)$?

These type of questions are all part of a larger and deeper one: *what is the message complexity of election in a ring* ? To answer this question, we need to establish a *lower bound*, a limit which no election protocol can improve upon, regardless of the amount and cleverness of the design effort.

In this section we will see different bounds, some for unidirectional rings and others for bidirectional ones, depending on the amount of a priori knowledge the entities have about the ring. As we will see, in all cases, the lower bounds are all of the form $\Omega(n \log n)$. Thus, any further improvement can only be in the multiplicative constant.

3.8.1 Unidirectional rings

We want to know what is the number of messages that any election algorithm for unidirectional rings must transmit in the *worst case*. A subtler question is to determine the number of messages that any solution algorithm must transmit on the *average*; clearly, a lower bound on the average case is also a lower bound on the worst case[1].

We will establish a lower-bound under the standard assumptions of Connectivity and Total Reliability, plus Initial Distinct Values (required for election), and obviously *Ring*. We will actually establish the bound assuming that there is Message Ordering; this implies that, in systems without Message Ordering, the bound is at least as bad. The lower-bound will be established for *minimum-finding* protocols; because of the Initial Distinct Values restriction, every minimum-finding protocol is also an election protocol. Plus we know that with an additional n messages every election protocol becomes a minimum-finding protocol.

When a minimum-finding algorithm is executed in a ring of entities with distinct values, the total number of transmitted messages depends on two factors: communication delays and the assignment of initial values.

[1] the converse is not true !

Consider the unidirectional ring $\vec{R} = (x_0, x_1, , \ldots, x_{n-1})$; let $s_i = id(x_i)$ be the unique value assigned to x_i. The sequence $\mathbf{s} =< s_1, s_2, \ldots, s_n >$ thus describes the assignment of ids to the entities.

Denote by S the set of all such assignments. Given a ring R of size n and an assignment $\mathbf{s} \in S$ of n ids, we will say that $\vec{R}$ is labelled by $\mathbf{s}$, and denote it by $\vec{R}(\mathbf{s})$.

Let A be a minimum-finding protocol under the restrictions stated above. Consider the executions of A started simultaneously by all entities, and their cost. The average and the worst-case cost of these executions are possibly better but surely not worse than the average and the worst-case costs, respectively, over all possible executions; thus, if we find them, they will give us a lower bound.

Call *global state* of an entity x at time t, the content of all its local registers and variables at time t. As we know, the entities are *event-driven*. This means that, for a fixed set of rules A, their next global state will depend solely on the current one and on what event has occurred. In our case, once the execution of A is started, the only external events are the arrival of messages.

During an action, an entity might send one or more message to its only out-neighbour; if it is more than one, we can "bundle" them together since they are all sent within the same action (i.e., before any new message is received). Thus, we assume that in A, only one message is sent in the execution of an action by an entity.

Associate to each message all the "history" of that message. That is, with each message M we associate a sequence of values, called *trace*, as follows: (1) if the sender has id s_i and has no previously received any message, the trace will be just $< s_i >$. (2) if the sender has id s_i and its last message previously received has trace $< l_1, \ldots, l_{k-1} >$, $k > 1$, the trace will be $< l_1, \ldots, l_{k-1}, s_i >$, which has length k.

Thus, a message M with trace $< s_i, s_{i+1} \ldots, s_{i+k} >$ indicates that: a message was originally sent by entity x_i; as a reaction, the neighbour x_{i+1}, sent a message; as a reaction, the neighbour x_{i+2} sent a message; $\ldots$; as a reaction, x_{i+k} sent the current message M.

IMPORTANT. Note that, because of our two assumptions (simultaneous start by all entities, and only one message per action), messages are uniquely described by their associated trace.

We will denote by **ab** the concatenation of two sequences **a** and **b**. If $\mathbf{d} = \mathbf{abc}$, then **a**, **b**, and **c** are called *subsequences* of **d**; in particular, each of **a**, **ab** and **abc** will be called a *prefix* of **d**; each of **c**, **bc** and **abc** will be called a *suffix* of **d**. Given a sequence **a**, will denote by $len(\mathbf{a})$ the length of **a**, and by $C(\mathbf{a})$ the set of cyclic permutations of **a**; clearly, $|C(\mathbf{a})| = len(\mathbf{a})$.

Example. If $\mathbf{d} =< 2, 15, 9, 27 >$, then: $len(\mathbf{d}) = 4$; the subsequences $< 2 >$, $< 2, 15 >$, $< 2, 15, 9 >$ and $< 2, 15, 9, 27 >$ are prefixes; the sequences $< 27 >$, $< 9, 27 >$, $< 15, 9, 27 >$, and $< 2, 15, 9, 27 >$ are suffixes; and $C(\mathbf{d}) = \{< 2, 15, 9, 27 >, < 15, 9, 27, 2 >, < 9, 27, 2, 15 >, < 27, 2, 15, 9 >\}$.

The key point to understand is the following: if in two different rings, e.g. in $\vec{R}(\mathbf{a})$ and in

$\vec{R}(\mathbf{b})$, an entity executing A happens to have the same global state, and it receives the same message, then it will perform the same action in both cases, and the next global state will be the same in both executions.

Let us use this point.

Lemma 3.1 *Let* $\mathbf{a}$ *and* $\mathbf{b}$ *both contain* $\mathbf{c}$ *as a subsequence. If a message with trace* $\mathbf{c}$ *is sent in an execution of* A *on* $\vec{R}(\mathbf{a})$*, then* $\mathbf{c}$ *is sent in an execution of* A *on* $\vec{R}(\mathbf{b})$*.*

Proof. Assume that a message with trace $\mathbf{c} =< s_i, \ldots, s_{i+k} >$ is sent when executing A on $\vec{R}(\mathbf{a})$. This means that, when entity x_i started the trace, it had not received any other message, and so, the transmission of this message was part of its initial "spontaneous" action; since the nature of this action depends only on A, x_i will send the message both in $\vec{R}(\mathbf{a})$ and in $\vec{R}(\mathbf{b})$. This message was the first and only message x_{i+1} received from x_i both in $\vec{R}(\mathbf{a})$ and in $\vec{R}(\mathbf{b})$; in other words, its global state until it received the message with trace starting with $< s_i >$ was the same in both rings; hence, it will send the same message with trace $< s_i, s_{i+1} >$ to x_{i+2} in both situations. In general, between the start of the algorithm and the arrival of a message with trace $< s_i, \ldots, s_{j-1} >$, entity x_j with id s_j, $i < j \leq i + k$ is in the same global state, and sends and receives the same message in both $\vec{R}(\mathbf{a})$ and $\vec{R}(\mathbf{b})$; thus, it will send a message with trace $< s_i, \ldots, s_{j-1}, s_j >$ regardless of whether the input sequence is $\mathbf{a}$ or $\mathbf{b}$.

Thus, if an execution of A in $\vec{R}(\mathbf{a})$ has a message with trace $\mathbf{c}$, then there is an execution of A in $\vec{R}(\mathbf{b})$ which has a message with trace $\mathbf{c}$. ■

In other words, if $\vec{R}(\mathbf{a})$ and $\vec{R}(\mathbf{b})$ have a common segment $\mathbf{c}$ (i.e., a consecutive group of $len(\mathbf{c})$ entities in $\vec{R}(\mathbf{a})$ has the same ids as a consecutive group of entities in $\vec{R}(\mathbf{b})$) the entity at the end of the segment cannot distinguish between the two rings when it sends the message with trace $\mathbf{c}$.

Since different assignments of values to rings may lead to different results (i.e., different minimum value), the protocol A must allow the entities to distinguish between those assignments. As we will see, this will be the reason $\Omega(n \log n)$ messages are needed. To prove it, we will consider a set of assignments on rings which makes distinguishing among them "expensive" for the algorithm.

A set $E \subseteq S$ of assignments of values is called *exhaustive* if it has the following two properties:

1. (*Prefix Property*) For every sequence belonging to E, its nonempty prefixes also belongs to E; that is, if $\mathbf{ab} \in E$ and $len(\mathbf{a}) \geq 1$, then $\mathbf{a} \in E$.

2. (*Cyclic Permutation Property*) Whether an assignment of values s belongs or not to E, at least one of its cyclic permutations belongs to E; that is, if $s \in S$, then $C(s) \cap E \neq \phi$

Lemma 3.2 *A has an exhaustive set $E(A) \subseteq S$.*

Proof. Define $E(A)$ to be the set of all the arrangements $\mathbf{s}\in S$ such that a message with trace $\mathbf{s}$ is sent in the execution of A in $\vec{R}(\mathbf{s})$. To prove that this set is exhaustive, we need to show that the cycle permutation property and the prefix property hold.

To show that the prefix property is satisfied, choose an arbitrary $\mathbf{s}=\mathbf{ab} \in E(A)$ with $len(\mathbf{a}) \geq 1$; by definition of $E(A)$, there will be a message with trace $\mathbf{ab}$ when executing A in $\vec{R}(\mathbf{ab})$; this means that in $\vec{R}(\mathbf{ab})$ there will also be a message with trace $\mathbf{a}$. Consider now the (smaller) ring $\vec{R}(\mathbf{a})$; since $\mathbf{a}$ is a subsequence of both $\mathbf{ab}$ and (obviously) $\mathbf{a}$, and there was a message with that trace in $\vec{R}(\mathbf{ab})$, by Lemma 3.1 there will be a message with trace $\mathbf{a}$ also in $\vec{R}(\mathbf{a})$; but this means that $\mathbf{a}\in E(A)$. In other words, the suffix property holds.

To show that the cyclic permutation property is satisfied, choose an arbitrary $\mathbf{s}=< s_1, \ldots, s_k >\in S$ and consider $\vec{R}(\mathbf{s})$. At least one entity must receive a message with a trace of length k, otherwise the minimum value could not have been determined; then $\mathbf{t}$ is a cyclic permutation of $\mathbf{s}$. Furthermore, since $\mathbf{t}$ is a trace in $\vec{R}(\mathbf{t})$, $\mathbf{t} \in E(A)$. Summarizing, $\mathbf{t} \in E(A) \cup S(\mathbf{s})$. In other words, the cyclic permutation property holds. ■

Now we are going to measure how expensive it is for the algorithm A to distinguish between the elements of $E(A)$.

Let $m(\mathbf{s}, E)$ be the number of sequences in $E \subseteq S$, which are prefixes of some cyclic permutation of $\mathbf{s} \in S$, and $m_k(\mathbf{s}, E)$ denote the number of those which are of length $k > 1$.

Lemma 3.3 *The execution of A in $\vec{R}(\mathbf{s})$ costs at least $m(\mathbf{s}, E(A))$ messages.*

Proof. Let $\mathbf{t}\in E(A)$ be the prefix of some $\mathbf{r}\in C(\mathbf{s})$. That is, a message with trace $\mathbf{t}$ is sent in $\vec{R}(\mathbf{t})$ and, because of Lemma 3.1, a message with trace $\mathbf{t}$ is sent also in $\vec{R}(\mathbf{r})$; since $\mathbf{r}\in C(\mathbf{s})$, then a message with trace $\mathbf{t}$ is sent also in $\vec{R}(\mathbf{r})$. That is, for each prefix $\mathbf{t}\in E(A)$ of a cyclic permutation of $\mathbf{s}$ there will be a message sent with trace $\mathbf{t}$. The number of such prefixes $\mathbf{t}$ is by definition $m(\mathbf{s}, E(A))$. ■

Let $I = \{s_1, s_2, \ldots, s_n\}$ be the set of ids, $Perm(I)$ be the set of permutations of I. Assuming that all $n!$ permutations in $Perm(I)$ are equally likely, the average number $ave_A(I)$ of messages sent by A in the rings labelled by I, will be the average message cost of A among the rings $\vec{R}(\mathbf{s})$, where $\mathbf{s}\in Perm(I)$. By Lemma 3.3, this means:

$$ave_A(I) \geq \tfrac{1}{n!}\textstyle\sum_{\mathbf{s}\in Perm(I)}\, m(\mathbf{s}, E(A))$$

By definition of $m_k(\mathbf{s}, E(A))$, we have

$$ave_A(I) \geq \tfrac{1}{n!}\textstyle\sum_{\mathbf{s}\in Perm(I)}\sum_{k=1}^{n} m_k(\mathbf{s}, E(A)) \;= \tfrac{1}{n!}\sum_{k=1}^{n}\;\sum_{\mathbf{s}\in Perm(I)} m_k(\mathbf{s}, E(A))$$

We need to determine what $\sum_{\mathbf{s}\in Perm(I)} m_k(\mathbf{s}, E(A))$ is. Fix k and $\mathbf{s}\in Perm(I)$. Each cyclic permutations $C(\mathbf{s})$ of $\mathbf{s}$ has only one prefix of length k. In total, there are n prefixes of of length k among all the cyclic permutations of $\mathbf{s}\in Perm(I)$. Since there are $n!$ elements in

$Perm(I)$, there are $n!\ n$ instances of such prefixes for a fixed k. These $n!\ n$ prefixes can be partitioned in groups G_j^k of size k, by putting together all the cyclic permutations of the same sequence; there will be $q = \frac{n!\ n}{k}$ such groups. Since $E(A)$ is exhaustive, by the cyclic permutation property, the set $E(A)$ intersects each group, that is $|E(A) \cup G_j^k| \geq 1$.

$$\sum_{s \in Perm(I)} m_k(\mathbf{s}, E(A)) \geq \sum_{j=1}^{q} |E(A) \cup G_j^k| \geq \frac{n!n}{k}$$

Thus,

$$ave_A(I) \geq \frac{1}{n!} \sum_{k=1}^{n} \frac{n!n}{k} \geq n \sum_{k=1}^{n} \frac{1}{k} = nH_n$$

where H_n is the n-th harmonic number. This lower bound on the average case, is also a lower bound on the number $worst_A(I)$ of messages sent by A in the worst case in the rings labelled by I:

$$worst_A(I) \geq ave_A(I) \geq nH_n \approx 0.69\ n \log n + O(n) \tag{28}$$

This result states that $\Omega(n\ logn)$ messages are needed in the worst case by *any* solution protocol (the bound is true for every A), even if there is Message Ordering. Thus, any improvement we can hope to obtain by clever design will at most reduce the constant; in any case, the constant can not be smaller than .69. Also, we can not expect to design election protocols that might have a bad worst case but cost dramatically less on the average. In fact, $\Omega(n\ logn)$ messages are needed on the average by any protocol.

Notice that the lower bound we have established can be achieved ! In fact, protocol *AsFar* requires on the average nH_n messages (Theorem 3.1). In other words, protocol *AsFar* is *optimal* on the average.

If the entities know n, it might be possible to develop better protocols; so far *none is known.* In any case (Exercise 9.45):

$$worst_A(I|n\ known) \geq ave_A(I|n\ known) \geq (\frac{1}{4} - \epsilon)\ n \log n \tag{29}$$

That is, even with the additional knowledge of n, the improvement can only be in the constant.

3.8.2 Bidirectional Rings

In bidirectional rings, the lower bound is slightly different both in derivation and value (Exercise 9.46).

$$worst_A(I) \geq ave_A(I) \geq \frac{1}{2}\ nH_n \approx 0.345\ n \log n + O(n) \tag{30}$$

Again, if the entities know n, it might be possible to develop better protocols; so far, *none is known.* In any case (Exercise 9.47):

$$worst_A(I : n \ known) \geq ave_A(I : n \ known) \geq \frac{1}{2} n \log n \tag{31}$$

That is, even with this additional knowledge, the improvement can only be in the constant.

3.8.3 Practical and Theoretical Implications

The lower bounds we have discussed so far indicate that $\Omega(n \log n)$ messages are needed both in the worst case and on the average, regardless of whether the ring is unidirectional or bidirectional, and whether n is known or not. The only difference between these case will be in the constant. In the previous sections we have seen several protocols that use $O(n \log n)$ messages in the worst case (and are thus optimal); their cost provide us with upper bounds on the complexity of leader election in a ring.

If we compare the best upper and lower bounds for unidirectional rings with those for bidirectional rings, we notice the existence of a very surprising situation: the bounds for unidirectional rings are "better" than those for bidirectional ones: the upper-bound is smaller and the lower bound is bigger (see Figures 34 and 33). This fact has strange implications: as far as the electing a leader in a ring is concerned, unidirectional rings seem to be better systems than bidirectional ones; which in turn implies that practically

half duplex links are better than full duplex links !

This is clearly counter-intuitive: in terms of communication hardware, bidirectional links are clearly more powerful than half-duplex links. On the other hand, the bounds are quite clear: election protocols for unidirectional rings are more efficient than those for bidirectional ones.

A natural reaction to this strange status of affairs is to suggest the use in bidirectional rings of unidirectional protocols; after all, with bidirectional links we can send in both directions, "left" and "right", so we can just decide to use only one, say "right". Unfortunately this argument is based on the hidden assumption that the bidirectional ring is also *oriented*; that is, "right" means the same to all processors. In other words, it assumes that the labelling of the port numbers, which is purely local, is actually *globally consistent*.

This explains why we cannot use the (more efficient) unidirectional protocol in a generic bidirectional ring. But why should we do better in unidirectional rings ?

The answer is interesting: *in a unidirectional ring, there is orientation*: each entity has only one out-neighbour so there is no ambiguity so to where to send a message. In other words, we have discovered an important principle of the nature of distributed computing

global consistency is more important than hardware communication power.

This principle is quite general. In the case of rings, the difference is not much, just in the multiplicative constant. As we will see in other topologies, this difference can actually be dramatic.

bidirectional protocol	*worst case*	*average*	*notes*
All the Way	n^2	n^2	
AsFar	n^2	$0.69n \log n + O(n)$	
ProbAsFar	n^2	$0.49n \log n + O(n)$	
Control	$6.31n \log n + O(n)$		
Stages	$2n \log n + O(n)$		
StagesFbk	$1.89n \log n + O(n)$		
Alternate	$1.44n \log n + O(n)$		oriented ring
BiMinMax	$1.44n \log n + O(n)$		
lowerbound		$0.35n \log n + O(n)$	$n = 2^p$ known
lowerbound		$0.25n \log n + O(n)$	

Figure 33: Summary of bounds for bidirectional rings.

unidirectional protocol	*worst case*	*average*	*notes*
All the Way	n^2	n^2	
AsFar	n^2	$0.69n \log n + O(n)$	
UniStages	$2n \log n + O(n)$		
UniAlternate	$1.44n \log n + O(n)$		
MinMax	$1.44n \log n + O(n)$		
MinMax+	$1.271n \log n + O(n)$		
lowerbound		$0.173n \log n + O(n)$	$n = 2^p$ known
lowerbound		$0.69n \log n + O(n)$	

Figure 34: Summary of bounds for unidirectional rings.

If the ring is both bidirectional and oriented, then we can clearly use any unidirectional protocol as well as any bidirectional one. The important question is whether in this case we can do better than that. That is, the quest is for a protocol for bidirectional oriented rings that

1. fully exploits the power of both full duplex links and orientation
2. cannot be used nor simulated in unidirectional rings, nor in general bidirectional ones; and
3. is more efficient than any unidirectional protocol or general bidirectional one.

We have seen a protocol for oriented rings, *Alternate*; however, it can be simulated in unidirectional rings (protocol *UniAlternate*). To date, no protocol with such properties is known. It is not even known whether it can exist. (Problem 9.7)

3.9 Summary and Lessons

We have examined the design of several protocols for leader election in ring networks, and analyzed the effects that design decisions have had on the costs.

When developing the election protocols, we have introduced some key strategies that are quite general in nature and, thus, can be used for different problems and for different networks. Among them: the idea of *electoral stages* and the concept of *controlled distances*. We have also employed ideas and tools, e.g. *feedback* and *notification*, already developed for other problems.

In terms of costs, we have seen that $\Theta(n \log n)$ messages will be used both in the worst case and on the average, regardless of whether the ring is unidirectional or bidirectional, oriented or unoriented, and n is known or not. The only difference is in the multiplicative constant. The bounds are summarized in Figures 34 and 33. As a consequence of these bounds, we have seen that orientation of the ring is, so far, more powerful than presence of bidirectional links.

Both ring networks and tree networks have very sparse topologies: $m = n - 1$ in trees and $m = n$ in rings. In particular, if we remove any single link from a ring we obtain a tree. Still, electing a leader costs $\Theta(n \log n)$ in rings but only $\Theta(n)$ in trees. The reason for such a drastic complexity difference has to be found not in the number of links but instead on the properties of the topological structure of the two types of networks. In a tree there is a high level of *asymmetry*: we have two types of nodes, internal nodes and leaves; it is by exploiting such asymmetry that election can be performed in a linear number of messages. On the contrary, a ring is a highly symmetrical structure, where every node is indistinguishable from another. Consider that the election task is really a task of breaking symmetry: we want one entity to become different from all others. The entities already have a behavioral symmetry: they all have the same set of rules, the same initial state, and potentially they are all initiators. Thus, the structural symmetry of the ring topology only make the solution to the problem more difficult and more expensive. This observation reflects a more general principle: as far as election is concerned, structural asymmetry is to the protocol designer's advantage; on the contrary, *the presence of structural symmetry is an obstacle for the protocol designer*.

4 Election in Mesh Networks

Mesh networks constitute a large class of architectures that includes *meshes* and *tori*; this class is popular especially for parallel systems, redundant memory systems, and interconnection networks. These networks, like trees and rings, are sparse: $m = O(n)$. Using our experience with trees and rings, we will now approach the election problem in such networks. Unless otherwise stated, we will consider Bidirectional Links.

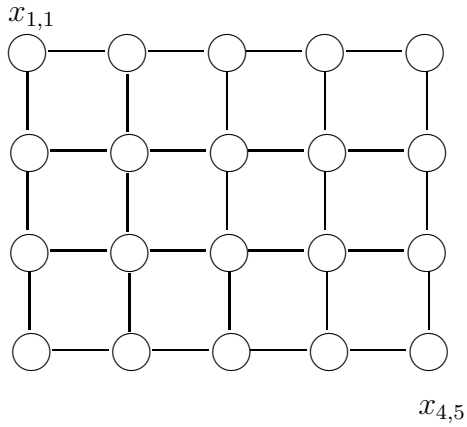


Figure 35: Mesh of dimension 4×5.

4.1 Meshes

A *mesh* M of dimensions $a \times b$ has $n = a \times b$ nodes, $x_{i,j} (1 \leq i \leq a, 1 \leq j \leq b)$. Each node $x_{i,j}$ is connected to $x_{i-1,j}$, $x_{i,j-1}$, $x_{i+1,j}$, $x_{i,j+1}$ if they exist; let us stress that these names are used for descriptive purposes only and are not known to the entities. The total number of links is thus $m = a(b-1) + b(a-1) = 2ab - a - b$ (see Fig. 35).

Observe that, in a mesh, we have three types of nodes: *corner* (entities with only two neighbours), *border* (entities with three neighbours), and *interior* (with four neighbours) nodes. In particular, there are 4 corner nodes, $2(a+b-4)$ border nodes, and $n-2(a+b-2)$ interior nodes.

4.1.1 Unoriented Mesh

The asymmetry of the mesh can be exploited to our advantage when electing a leader: since it does not matter which entity becomes leader, we can elect one of the four corner nodes. In this way, the problem of choosing a leader between (possibly) n nodes is reduced to the problem of choosing a leader among the four corner nodes. Recall that any number of nodes can start (each unaware of when and where the others will start, if at all); thus, to achieve our goal, we need to design a protocol that first of all makes the corners aware of the election process (they might not be initiators at all), and then performs the election among them.

The first step, to make the corners aware, can be performed doing a *wake up* of all entities. When an entity wakes up (spontaneously if it is an initiator, upon receiving a wake-up message otherwise), its subsequent actions will depend on whether it is a corner, a border or an interior nodes.

In particular, the four corners will become awake and can start the actual election process.

Observe the following interesting property of a mesh: if we consider only the border and corner nodes and the links between them, they form a ring network. We can thus elect a

leader among the corners by using a election protocol for rings: the corners will be the only *candidates*, the borders will act as relayers (*defeated* nodes). When one of the corner nodes is elected, it will notify all other entities of termination.

Summarizing the process will consist of:
(1) Wake-up, started by the initiators;
(2) Election (on outer ring), among the corners;
(3) Notification (i.e., broadcast) started by the leader.

Wake up is straightforward. Each of the $k_\star$ initiators will send a wake-up to all its neighbours; a non-initiator will receive the wake-up message from a neighbour and forward it to all its other neighbours (no more than 3); hence the number of messages (Exercise 9.48) will be no more than

$$3n + k_\star$$

The election on the outer ring requires a little more attention.

First of all, we must choose which ring protocol we will use; clearly, the selection is among the efficient ones we have discussed at great length in the preceding sections.

Then we must ensure that the messages of the ring election protocol are correctly forwarded along the links of the outer ring.

Let us use protocol *Stages*, and consider the first stage. According to the protocol, each *candidate* (in our case, a corner node) sends a message containing its value in both directions in the ring; each *defeated* entity (in our case, a border node) will forward the message along the (outer) ring.

Thus, in the mesh, each corner node will send a message to the only two neighbours. A border node y, however, has *three* neighbours, of which only two are in the outer ring; when y receives the message, it does not know to which of the other two ports it must forward the message. What we will do is simple; since we do not know to which, we will forward to *both*: one will be along the ring and proceed safely, the other will instead reach an interior node z; when the interior node z receives such an election message, it will reply to the border node y "I am in the interior", so no subsequent election messages are sent to it. Actually, it is possible to avoid those replies without affecting the correctness (Exercise 9.50).

In *Stages* the number of *candidates* is at least halved every time. This means that, after the second stage, one of the corners will determine that it has the smallest id among the four candidates and will become *leader*.

Each stage requires $2n'$ messages, where $n' = 2(a+b-2)$ is the dimension of the outer ring. An additional $2(a+b-4)$ are unknowingly sent by the border to the interior in the first stage; there are also the $2(a+b-4)$ replies from those interior nodes, that however can be avoided (Exercise 9.50). Hence, the number of messages for the election process will be at most

$$4(a+b-2) + 2(a+b-4) = 6(a+b) - 16.$$

IMPORTANT. Notice that in a *square* mesh (i.e., a=b), this means that the election process proper can be achieved in $O(\sqrt{n})$ messages !

Broadcasting the notification can be performed using *Flood*, which will require less than $3n$ messages since it is started by a corner. Actually, with care, we can ensure that less than $2n$ messages are sent in total (Exercise 9.49).

Thus in total, the protocol *ElectMesh* we have designed will have cost:

$$\mathbf{M}[ElectMesh] \leq 6(a+b) + 5n + k_\star - 16 \tag{32}$$

NOTES.
The most expensive operation is to wake-up the nodes.
In a *square* mesh (i.e., a=b), the election process proper can be achieved in $O(\sqrt{n})$ messages !

Hacking:
With a simple modification to the protocol, it is possible to save an additional $2(a+b-4)$ messages (Exercise 9.51), achieving a cost of at most

$$4(a+b) + 5n + k_\star - 32.$$

4.1.2 Oriented Mesh

A mesh is called *oriented* if the port numbers are the traditional *compass* labels (*North, South, East, West*) assigned in a globally consistent way. This assignment of labels has many important properties, in particular one called *sense of direction* which we will discuss later in much detail. These properties can be exploited to obtain efficient solutions to problems such as broadcast and traversal (Problems 9.52 and 9.53). For the purposes of election,

in an oriented mesh, it is trivial to identify a unique node.

For example, there is only one corner with link labels "South" and "West". Thus, to elect a leader in an oriented mesh, we must just ensure that that unique node knows that it must become *leader*.

In other words, the only part needed is a Wake-up: upon becoming awake, and participating in the wakeup process, an entity can immediately become *follower* or *leader*.

Notice that, in an oriented mesh, we can exploit the structure of the mesh and the orientation to perform a WakeUp with fewer than $2n$ messages (Problem 9.54).

Complexity
These results mean that, regardless of whether the mesh is oriented or not, a leader can be elected with $O(n)$ messages, the difference being solely in the multiplicative constant. Since no election protocol for any topology can use fewer than n messages, we have

Lemma 4.1 $\mathcal{M}$*(***Elect***/***RI** *; Mesh)* $= \Theta(n)$

4.2 Tori

Informally, the *torus* is a mesh with "wrap-around" links which transform it into a *regular* graph: every node will have exactly four neighbours.

A torus of dimensions $a \times b$ has $n = ab$ nodes $v_{i,j}$ $(0 \leq i \leq a-1, 0 \leq j \leq b-1)$; each node $v_{i,j}$ is connected to four nodes $v_{i,j+1}$, $v_{i,j-1}$, $v_{i+1,j}$, and $v_{i-1,j}$; all the operations on the first index are *modulo* a, while those on the second index are *modulo* b. For an example, see Figure 36.

In the following we will focus on *square* tori (i.e., where $a = b$).

4.2.1 Oriented Torus

We will first develop an election protocol assuming that there is the *compass* labelling (i.e., the links are consistently labelled *North, South, East, West* and the dimensions are known); we will then see how to solve the problem also when the labels are arbitrary. A torus with such a labelling is said to be *oriented*.

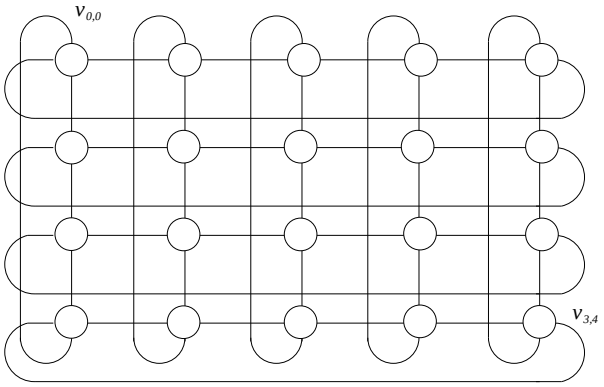


Figure 36: Torus of dimension 4×5.

In designing the election protocol, we will use the idea of *electoral stages* developed originally for ring networks, and we will use the *defeated* nodes in an active way. We will also employ a new idea, *marking of territory*.

(I) In stage i, each *candidate* x must "mark" the boundary of a territory T_i (a $d_i \times d_i$ region of the torus), where $d_i = \alpha^i$ for some fixed constant $\alpha > 1$; initially the territory is just the single *candidate* node. The marking is done by originating a "Marking" message (with x's value) that will travel to distance[2] d_i first $North$, then $East$, then $South$, and finally $West$ to return to x.

[2]Distances include the starting node.

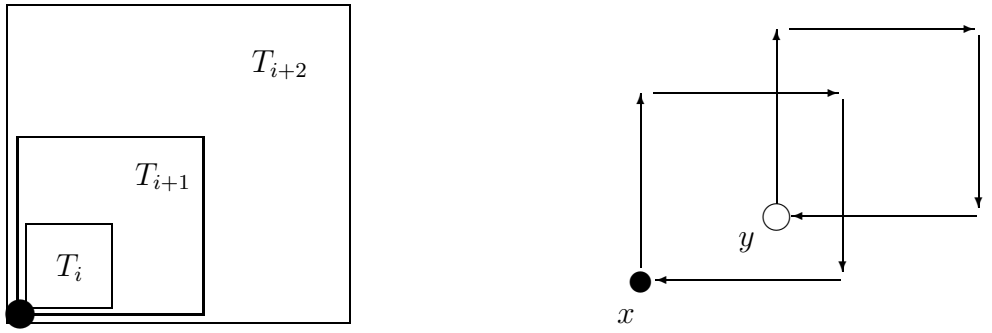


Figure 37: Marking the territory. If the territories of two *candidates* intersect, one of them will see the marking of the other.

A very important fact is that, if the territory of two *candidates* have some elements in common, the Marking message of at least one of them will encounter the marking of the other (Figure 37).

(II) If the "Marking" message of x does not encounters any other marking of the same stage, x survives this stage, enters stage $i+1$ and starts the marking of a larger territory T_{i+1} .

(III) If the "Marking" message arrives at a node w already marked by another *candidate* y in the same stage, the following will occur.

1. If y has a larger Id, the Marking message will continue to mark the boundary, setting a boolean variable SawLarger to *true*.
2. If the Id of y is instead smaller, then w will terminate the "Marking" message from x; it will then originate a message "SeenbyLarger(x, i)" that will travel along the boundary of y' territory.

If *candidate* x receives both its Marking message with SawLarger=*true* and a "SeenbyLarger" message, x survives this stage, enters stage $i+1$ and starts the marking of a larger territory T_{i+1}.

Summarizing, for a *candidate* x to survive, it is necessary that it receives its "Marking" message back. If SawLarger=*false*, then that suffices; if SawLarger=*true*, x must also receive a "SeenbyLarger" message.

Note that, if x receives a "SeenbyLarger(z, i)" message, then z did not finish marking its boundary; thus z does not survives this stage. In other words, if x survives, either its message found no other markings, or at least another *candidate* does not survive.

(IV) A relay node w might receive several "Marking" messages from different *candidates* in

the same stage. It will only be part of the boundary of the territory of the *candidate* with the smallest id.

This means that if w was part of the boundary of some *candidate* x and now becomes part of the boundary of y, a subsequent "SeenbyLarger" message intended for x will be sent along the boundary of y. This is necessary for correctness. To keep the number of messages small, we will also limit the number of "SeenbyLarger" messages sent by a relayer.

(V) A relay node will only forward one "SeenbyLarger" message.

The algorithm continues in this way until $d_i \geq \sqrt{n}$. In this case, a *candidate* will receive its "Marking" message from $South$ instead of $East$, due to the "wrap around" in the torus; it then sends the message directly $East$, and will wait for it to arrive from $West$.

(VI) When a wrap-around is detected (receive its "Marking" message from $South$ rather than $East$), a *candidate* x sends the message directly $East$, and waits for it to arrive from $West$.

If it survives, in all subsequent stages the marking becomes simpler.

(VII) In every stage after wrap-around, a *candidate* x sends its "Marking" message first $North$, and waits to receive it from $South$, then it sends it $East$, and waits for it to arrive from $West$.

The situation where there is only one *candidate* left will be for sure reached a constant number p of stages after the wrap-around occurs, as we will see later.

(VIII) If a *candidate* x survives p stages after wrap-around, it will become *leader* and notify all other entities of termination.

Let us now discuss the correctness and cost of the algorithm, Protocol *MarkBoundary*, we have just described.

Correctness and Cost

For the correctness, we need to show *progress* - i.e., at least one *candidate* survives each stage of the algorithm- and *termination* -i.e., p stages after wrap-around there will be only one *candidate* left.

Let us discuss progress first. A *candidate*, whose "Marking" message does not encounter any other boundary, will survive this stage; so the only problem is if, in a stage, every "Marking" message encounters another *candidate*'s boundary, and somehow none of them advances. We

must show that this can not happen. In fact, if every "Marking" message encounters another *candidate*'s boundary, the one with the largest Id will encounter a smaller Id; the *candidate* with this smaller Id will go onto the next stage unless its message encountered the boundary with an even smaller id, and so on; however, the message of the *candidate* with the smallest Id can not encounter a larger Id (because it is the smallest) and thus that entity would survive this stage.

For termination, the number of *candidates* does decrease overall, but not in a simple way. However, it is possible to bound the maximum number of *candidates* in each stage, and that bound strictly decreases. Let n_i to be the maximum number of *candidates* in stage i. Up until wrap around, there are two types of survivors: (a) those entities whose message did not encounter any border, and (b) those whose message encountered a border with a larger id, and whose border was encountered by a message with a larger id. Let a_i denote the number of the first type of survivors; clearly $a_i \leq n/d_i^2$. The number of the second type will be at most $(n_i - a_i)/2$ since each defeated one can cause at most one *candidate* to survive. Thus

$$n_{i+1} \leq a_i + (n_i - a_i)/2 = (n_i + a_i)/2 \leq (n_i + \tfrac{n}{d_i^2})/2$$

Since $d_i = \alpha^i$ is increasing each stage, the upper bound n_i on the number of *candidates* is decreasing. Solving the recurrence relation gives

$$n_{i+1} \leq n/\alpha^{2i}(2 - \alpha^2) \tag{33}$$

Wrap-around occurs when $\alpha^i \geq \sqrt{n}$; in that stage, only one *candidate* can can complete the marking of its boundary without encountering any markings, and at most half the remaining *candidates* will survive. So, the number of *candidates* surviving this stage is at most $(2 - \alpha^2)^{-1}$. In all subsequent stages, again only one *candidate* can can complete the marking without encountering any markings, and at most half the remaining *candidates* will survive. Hence, after

$$p > \lceil \log(2 - \alpha^2)^{-1} \rceil$$

additional stages for sure there will be only one *candidate* left. Thus, the protocol correctly terminates.

To determine the total number of messages, consider that, in stage i before wrap-around, each *candidate* causes at most $4d_i$ "Marking" messages to mark its boundary and another $4d_i$ "SeenbyLarger" messages, for a total of $8d_i = 8\alpha^i$ messages; as the number of *candidates* is at most as expressed by equation 33, the total number of messages in this pre-wrap-around stage will be at most

$$O(n\alpha^2 \ /(2 - \alpha^2)(\alpha - 1))$$

In each phase after wrap-around, there is only a constant number of *candidates*, each sending $O(\sqrt{n})$ messages. Since the number of such phases is constant, the total number of messages sent after wrap-around is $O(\sqrt{n})$.

Choosing $\alpha \approx 1.1795$ yields the desired bound

$$\mathbf{M}[MarkBoundary] = \Theta(n) \tag{34}$$

The preceding analysis ignores the fact that α^i is not an integer: the distance to travel must be rounded up and this has to taken into account in the analysis. However, the effect is not large and will just affect the low order terms of the cost (Exercise 9.55).

The algorithm as given is not very time efficient. In fact, the ideal time can be as bad as $O(n)$ (Exercise 9.56). The protocol can be however modified so that, without changing its message complexity requires no more than $O(\sqrt{n})$ time (Exercise 9.57).

The protocol we have described is tailored for square tori. If the torus is not square but rectangular with length l and width w ($l \leq w$), then the algorithm can be adapted to use $\Theta(n + l \log l/w)$ messages (Exercise 9.58).

4.2.2 Unoriented Torus

The algorithm we just described solved the problem of electing a leader in an oriented torus, e.g. among the buildings in Manhattan (well-known for its mesh-like design), by sending a messenger along east-west streets and north-south avenues, turning at the appropriate corner. Consider now the same problem when the streets have no signs and the entities have no compass.

Interestingly, the same strategy can be still used: a *candidate* needs to mark off a square; the orientation of the square is irrelevant. To be able to travel along a square, we just need to know how to (1) forward a message "in a straight line" and (2) make the "appropriate turn". We will discuss how to achieve each, separately.

Forwarding in a Straight Line. We first consider how to forward a message in the direction opposite to the one from which the message was received, without knowing the directions.

Consider an entity x, with its four incident links, and let a, b, c, d be the arbitrary port numbers associated with them; (see Figure 38); to forward a message in a straight line, x needs to determine that a and d are opposite, and so are b and c. This can be easily accomplishing by having each entity send its identity to each of its four neighbours, which will forward it to its three other neighbours; the entity will in turn acquire the identity and relative position of each entity at distance 2. As a result, x will know which the two pairs of opposite port numbers. In the example of Figure 38, x will receive the message originating from z via both port a and port b; it thus knows that a is not opposite to b. It also receives the message from y via ports a and c; thus x knows also that a is not opposite to c. Then, x can conclude that a is opposite to d.

It will then locally relabel one pair of opposite ports as $east, west$, and the other $north, south$; it does not matter which pair is chosen first.

Making the Appropriate Turn. As a result of the the previous operation, each entity

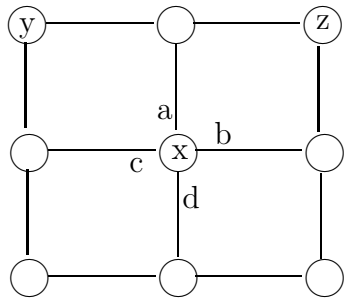


Figure 38: Even without a compass, x can determine which links are opposite.

x knows two *perpendicular* directions, but the naming (north,south) and (east,west) might not be consistent with the the one done by other entities. This can create problems when wanting to make a consistent turn.

Consider a message, originating by x which is travelling "south" (according to x's view of the torus); to continue to travel "south" can be easily accomplished since each entity knows how to forward a message in a straight line. At some point, according to the protocol, the message must turn, say to "east" (always according to x's view of the torus) and continue in that direction.

To achieve the turn correctly, we add a simple information, called *handrail*, to a message. The *handrail* is the id of the neighbour in the direction the message must turn, and the name of the direction. In the example of Figure 38, if x is sending a message *south* which must then turn *east*, the handrail in the message will be the id of its eastern neighbour q plus the direction "east". Because every entity knows the ids and the relative position of all the entities within distance 2, when y receives this message with the handrail from x, it can determine what x means by "east", and thus in which direction the message must turn (when the algorithm prescribes it).

Summarizing, even without a compass, we can execute the protocol *TorusElect*, by adding the preprocessing phase, and including the handrail information in the messages.

The cost of the preprocessing is relatively small: each entity receives four messages for its immediate neighbours and 4×3 for entities at distances 2, for a total of $16n$ messages.

5 Election in Cube Networks

5.1 Oriented Hypercubes

The k-dimensional oriented *hypercube* H_k, which we have introduced in Section **??**, is a common interconnection network, consisting of $n = 2^k$ nodes, each with degree k; hence, in H_k there are $m = k2^{k-1} = O(n \log n)$ edges.

In an oriented hypercube H_k, the port numbers $1, 2, \ldots, k$ for the k edges incident on a node

x are called *dimensions*, and are assigned according to the "construction rules" specifying H_k (see Figure 39):
The oriented hypercube H_1 of dimension $k = 1$ is just a pair of nodes called (in binary) "0"and "1", connected by a link labelled "1" at both nodes.
The oriented hypercube H_k of dimension $k > 0$ is obtained by taking two hypercubes of dimension $k-1$, H and Q, and connecting the corresponding pairs of nodes (i.e., with the same name) with a link labelled k; the name of each node in H (resp. Q) is then modified by prefixing it with the bit 0 (resp., 1).
It is important to observe that these names are used only for descriptive purposes, are not related to the id of the entities, and are unknown to the entities.

We will solve the election problem in oriented hypercubes using the approach, *electoral stages*, we have developed for ring networks. The metaphor we will use is that of a *fencing tournament*: in a stage of the tournament, each *candidate*, called *duelist*, will be assigned another duelist, and each pair will have a *match*; as a result of the match, one duelist will be promoted to the next stage, the other excluded from further competition. In each stage only half of the duelists enter the next stage; at the end, there will be only one duelist that will become the *leader* and notify the others.

Deciding the outcome of a match is easy: the duelist with the smaller id will win; for reasons that will become evident later, we will have the defeated duelist remember the shortest path to the winning duelist.

The crucial and difficult parts are how pairs of opposite duelists are formed, and how a duelist finds its competitor. To understand how this can be done efficiently, we need to understand some structural properties of oriented hypercubes.

A basic property of an oriented hypercube is that, if we remove from H_k all the links with label greater than i, (i.e., consider only the first i dimensions), we are left with 2^{k-i} disjoint oriented hypercubes of dimension i; denote the collection of these smaller cubes by $H_{k:i}$ For example, removing the links with label 3 and 4 from H_4 will result into 4 disjoint oriented hypercubes of dimension 2 (see Figure 39(a,b)).

What we will do is to ensure that

(I) at the end of stage $i-1$ there will be only one duelist left in each of the oriented hypercubes of dimension $i-1$ of $H_{k:i-1}$.

So, for example, at the end of stage 2, we want to have only one duelist left in each of the four hypercubes of dimension 2. (see Figure 39(c)).

Another nice property of oriented hypercubes is that, if we add to $H_{k:i-1}$ the links labelled i (and, thus, construct $H_{k:i}$) the elements of $H_{k:i-1}$ will be grouped into pairs.
We can use this property to form the pairs of duelists in each stage of the tournament:

(II) a duelist x starting stage i will have as its opponent the duelist in the hypercube of dimension $i-1$ connected to x by the link labelled i.

Thus, in stage i, a duelist x will send a Match message to (and receive a Match message from) the duelist y in hypercube (of dimension $i-1$) that is on the other side of link i. The

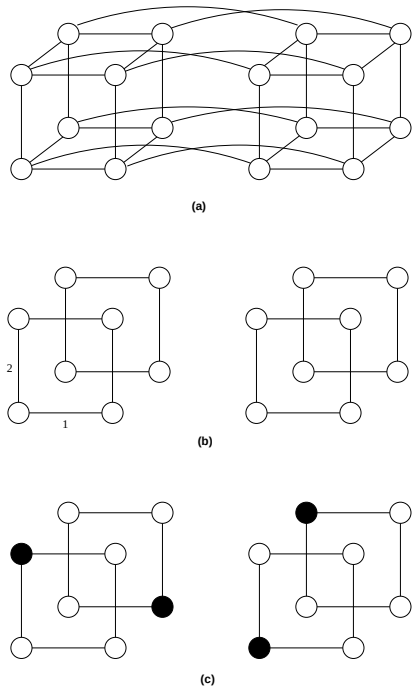


Figure 39: (a) the 4-dimensional hypercube H_4 and (b) the collection $H_{4:2}$ of 2-dimensional hypercubes obtained by removing the links with labels greater than 2. (c) Duelists (in black) at the end of stage 2.

Match message from x will contain the id $id(x)$ (as well as the path travelled so far) and will be sent across dimension i (i.e., the link with label i). The entity z on the other end of the link might however not be the duelist y and might not even know who (and where) y is (Figure 40) We need the Match message from x to reach its opponent y. We can obtain this by having z broadcast the message in its $(i-1)$-dimensional hypercube (e.g. using protocol *HyperFlood* presented in Section **??**); in this way we are sure that y will receive the message. Obviously, this approach is an expensive one (how expensive is determined in Exercise 9.59).

To solve this problem efficiently, we will use the following observation. If node z is not the duelist (i.e., $z \neq y$), node z was defeated in a previous stage, say $i_1 < i$; it knows the (shortest) path to the duelist z_{i_1} which defeated it in that stage, and can thus forward the message to it. Now, if $z_{i_1} = y$ then we are done: the message from x has arrived and the match can take place. Otherwise, in a similar way, z_{i_1} was defeated in some subsequent stage i_2, $i_1 < i_2 < i$; it thus knows the (shortest) path to the duelist z_{i_2} which defeated it in that stage, and can thus forward the message to it. In this way, the message from x will eventually reach y; the path information in the message is updated during its travel, so that y will know the dimensions traversed by the message from x to y in chronological order. The Match message from y will reach x with similar information.

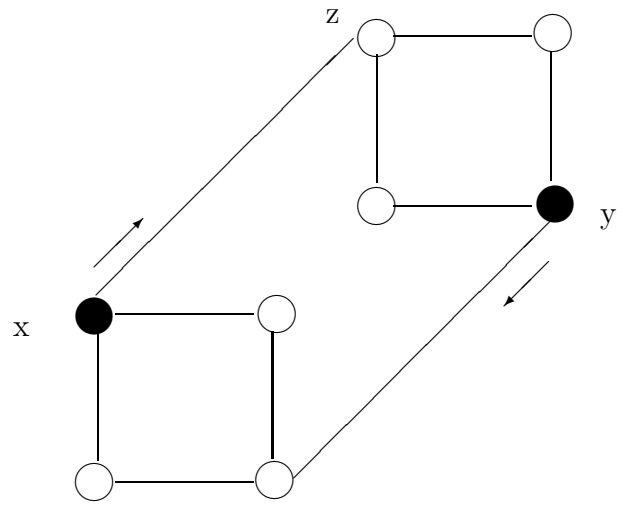


Figure 40: Each duelist (in black) sends a Match message that must reach its opponent.

The match between x and y will take place both at x and y; only one of them, say x, will enter stage $i+1$, while the other, y, is defeated.

From now on, if y receives a Match message, it will forward it to x; as mentioned before, we need this to be done on the shortest path. How can y (the defeated duelist) know the shortest path to x (the winner) ?

The Match message y received from x contained the labels of *a walk* to it, not necessarily the shortest path. Fortunately, it is easy to determine the shortcuts in any path using the properties of the labelling. Consider a sequence α of labels (with or without repetitions); remove from the sequence any pair of identical labels and sort the remaining ones, obtaining a *compressed* sequence $\overline{\alpha}$. For example, if $\alpha = \langle 231345212 \rangle$ then $\overline{\alpha} = \langle 245 \rangle$.
The important property is that, if we start from the same node x, the walk with labels α will lead to the same node y as the walk with labels $\overline{\alpha}$. The other important property is that $\overline{\alpha}$ actually corresponds to a shortest path between x and y. Thus, y needs only to compress the sequence contained in the Match message sent by x.
Important. We can perform the compression *while* the message is travelling from x to y; in this way the message will contain at most k labels.

Finally, we must consider the fact that, due to different transmission delays, it is likely that the computation in some parts of the hypercube is faster than in others. Thus, it may happen that a duelist x in stage i sends a Match message for its opponent, but the entities on the other side of dimension i are still in earlier stages.

So, it is possible that the message from x reaches a duelist y in an earlier stage $j < i$. What y should do with this message depends on future events that have nothing to do with the message: if y wins all matches in stages $j, j+1, \ldots, i-1$, then y is the opponent of x in stage i, and it is the destination of the message; on the other hand, if it loses one of them, it must forward the message to the winner of that match. In a sense, the message from x has

arrived "too soon"; so, what y will do is to delay the processing of this message until the "right" time, i.e. until it enters stage i or it becomes defeated.

Summarizing,

1. a *duelist* in stage i will send a Match message on the edge with label i;
2. when a *defeated* node receives a Match message, it will forward it to the winner of the match in which it was defeated;
3. when a *duelist* y in stage i receives a Match message from a *duelist* x in stage i: if $id(x) > id(y)$ then y will enter stage $i+1$, otherwise it will become *defeated* and compute the shortest path to x.
4. when a *duelist* y in stage j receives a Match message from a *duelist* x in stage $i > j$, y will enqueue the message and process it (as a newly arrived one) when it enters stage i or becomes *defeated*

The protocol terminates when a duelist wins the k-th stage. As we will see, when this happen, that duelist will be the only one left in the network.

The algorithm, protocol *HyperElect*, is shown in Figures 41 and 42.

NextDuelist denotes the (list of labels on the) path from a *defeated* node to the duelist which *defeated* it. The Match message contains: *(Id*, stage*, source*, dest*)*, where *Id** is the identity of the *duelist* x originating the message; *stage** is the stage of this match; *source** is (the list of labels on) the path from the duelist x to the entity currently processing the message; finally, *dest** is (the list of labels on) the path from the entity currently processing the message to a target entity (used to forward message by the shortest path between a defeated entity and its winner).

Given a list of labels *list*, the protocol uses the following functions:
- *first(list)* returns the first element of the list;
- $list \oplus i$ (resp. $\ominus$) updates the given path by adding (resp. eliminating) a label i to the list and compressing it.

To store the delayed messages, we use a set *Delayed* which will be kept *sorted* by stage number; for convenience, we also use a set *delay* of the corresponding stage numbers.

Correctness and termination of the protocol follow from the following fact (Exercise 9.61):

Lemma 5.1 *Let $id(x)$ be the smallest id in one of the hypercubes of dimension i in $H_{k:i}$. Then x is a duelist at the beginning of stage $i+1$.*

This means that, when $i = k$, there will be only one duelist left at the end of that stage; it will then become *leader* and notify the others so to ensure proper termination.

To determine the cost of the protocol, we need to determine the number of messages sent in a stage i. For a *defeated* entity z denote by $w(z)$ its opponent, (i.e., the one that won the match). For simplicity of notation, let $w^j(z) = w(w^{j-1}(z))$ where $w^0(z) = z$.

PROTOCOL HyperElect.

- States: $\mathcal{S}$ = {ASLEEP, DUELLIST, DEFEATED, FOLLOWER, LEADER}; $\mathcal{S}_{INIT}$ = {ASLEEP}; $\mathcal{S}_{TERM}$ = {FOLLOWER, LEADER}.
- Restrictions: **RI**$\cup\{OrientedHypercube\}$.

```
ASLEEP
      Spontaneously
      begin
           stage:= 1; delay:=0; value:= id(x);
           Source:= [stage];
           Dest:= [];
           send(``Match'', value, stage, Source, Dest) to 1;
           become DUELLIST;
      end

      Receiving(``Match'', value*, stage*, Source*, Dest*)
      begin
           stage:= 1; value:= id(x);
           Source:= [stage];
           Dest:= [];
           send(``Match'', value, stage, Source, Dest) to 1;
           become DUELLIST;
           if   stage* =stage then
                PROCESS_MESSAGE;
           else
                DELAY_MESSAGE;
           endif
      end


DUELLIST
      Receiving(``Match'', value*, stage*, Source*, Dest*)
      begin
           if   stage* =stage then
                PROCESS_MESSAGE;
           else
                DELAY_MESSAGE;
           endif
      end


DEFEATED
      Receiving(``Match'', value*, stage*, Source*, Dest*)
      begin
           if   Dest* = [] then Dest*:= NextDuelist; endif
           l:=first(Dest*); Dest:=Dest* ⊖l; Source:= Source* ⊕l;
           send(``Match'', value*, stage*, Source, Dest) to l;
      end

      Receiving(``Notify'')
      begin
           send ("Notify") to {l ∈ N(x) : l > sender};
           become FOLLOWER;
      end
```

Figure 41: Protocol *HyperElect*

```
Procedure PROCESS_MESSAGE
begin
     if  value* > value then
         if  stage* =k then
             send ("Notify") to N(x);
             become LEADER;
         else
             stage:= stage+1; Source:=[stage] ; dest:= [];
             send(‘‘Match’’, value, stage, Source, Dest) to stage;
             CHECK;
         endif
     else
         NextDuelist := Source;
         CHECK_ALL;
         become DEFEATED;
     endif
end


Procedure DELAY_MESSAGE
begin
     Delayed ⇐ (value*, stage*, Source*, Dest*);
     delay ⇐ stage*;
end


Procedure CHECK
begin
     if  Delayed ≠ ∅ then
         next:=Min{delay};
         if  next = stage then
             (value*, stage*, Source*, Dest*) ⇐ Delayed;
             delay:= delay-{next};
             PROCESS_MESSAGE
         endif
     endif
end


Procedure CHECK_ALL
begin
     while  Delayed ≠ ∅ do
            (value*, stage*, Source*, Dest*) ⇐ Delayed;
            if   Dest* [] then Dest*:= NextDuelist; endif
            l:=first(Dest*) ; Dest:=Dest* ⊖l ; Source:= Source* ⊕l
            send(‘‘Match’’, value*, stage*, Source, Dest) to l;
     endwhile
end
```

Figure 42: Procedures used by Protocol *HyperElect*

Consider an arbitrary $H \in H_{k:i-1}$; let y be the only *duelist* in H in stage i, and let z be the entity in H that receives first the Match message for y from its opponent. Entity z must send this message to y; it forwards the message (through the shortest path) to $w(z)$, which will forward it to $w(w(z)) = w^2(z)$, which will forward it to $w(w^2(z)) = w^3(z)$, and so on, until $w^t(z) = y$. There will be no more than i such "forward" points (i.e., $t \leq i$); since we are interested in the worst case, assume this to be the case. Thus the total cost will be the sum of all the distances between successive forward points, plus one (from x to z). Denote by $d(j-1, j)$ the distance between $w^{j-1}(z)$ and $w^j(z)$; clearly $d(j-1, j) \leq j$ (Exercise 9.60); then the total number of messages required for the Match message from a duelist x in stage i to reach its opposite y will be at most

$$L(i) = 1 + \sum_{j=1}^{i-1} d(j-1, j) = 1 + \sum_{j=1}^{i-1} j = 1 + \tfrac{i \cdot (i-1)}{2}.$$

Now we know how much does it cost for a Match message to reach its destination. What we need to determine is how many such messages are generated in each stage; in other words, we want to know the number n_i of *duelists* in stage i (since each will generate one such message). By Lemma 5.1, we know that at the beginning of stage i, there is only one *duelist* in each of the hypercubes $H \in H_{k:i-1}$; since there are exactly $\frac{n}{2^{i-1}} = 2^{k-i+1}$ such cubes,

$$n_i = 2^{k-i+1}.$$

Thus the total number of messages in stage i will be

$$n_i L(i) = 2^{k-i+1}(1 + \tfrac{i \cdot (i-1)}{2}).$$

and over all stages the total will be

$$\sum_{i=1}^{k} 2^{k-i+1}(1 + \tfrac{i \cdot (i-1)}{2}) = 2^k(\sum_{i=1}^{k} \tfrac{i}{2^{i-1}} + \sum_{i=1}^{k} \tfrac{i^2}{2^i} + \sum_{i=1}^{k} \tfrac{i}{2^i}) = 6\ 2^k - k^2 - 3k - 7.$$

Since $2^k = n$, and adding the $(n-1)$ messages to broadcast the termination, we have

$$\mathbf{M}[HyperElect] \leq 7n - (\log n)^2 - 3 \log n - 7 \tag{35}$$

That is, we can elect a leader in less than $7n$ messages ! This result should be contrasted with the fact that in a ring we need $\Omega(n \log n)$ messages.

As for the time complexity, it is not difficult to verify that protocol *HyperFlood* requires at most $O(\log^3 N)$ ideal time (Exercise 9.62).

Practical Considerations

The $O(n)$ message cost of protocol *HyperElect* is achieved by having the Match messages convey path information in addition to the usual id and stage number. In particular, the fields *Source* and *Dest* have been described as lists of labels; since we only send *compressed*

paths, *Source* and *Dest* contain at most $\log n$ labels each. So it would appear that the protocol requires "long" messages.

We will now see that, in practice, each list only requires $\log n$ bits (i.e., the cost of a counter).

Examine a compressed sequence of edge labels $\overline{\alpha}$ in H_k (e.g., $\overline{\alpha} = \langle 1457 \rangle$ in H_8); since the sequence is compressed, there are no repetitions. The elements in the sequence are a subset of the integers between 1 and k; thus $\overline{\alpha}$ can be represented as a binary string $\langle b_1, b_2, \ldots, b_k \rangle$ where each bit $b_j = 1$ if and only if j is in $\overline{\alpha}$. Thus, the list $\overline{\alpha} = \langle 1457 \rangle$ in H_8 is uniquely represented as $\langle 10011010 \rangle$. Thus, each of *Source* and *Dest* will be just a $k = \log n$ bits variable.

This also implies that the cost in terms of bits of the protocol will be no more than

$$\mathbf{B}[HyperElect] \leq 7n(\log \mathbf{id} + 2\log n + \log\log n) \tag{36}$$

where the $\log\log n$ component is to account for the *stage* field.

5.2 Unoriented Hypercubes

Hypercubes with arbitrary labellings obviously do not have the properties of oriented hypercubes. It is still possible to take advantage of the highly regular structure of hypercubes to do better than in ring networks. In fact, (Problem 9.8):

Lemma 5.2 $\mathcal{M}$*(***Elect***/***RI** *; Hypercube)* $\leq O(n \log\log n)$

To date, it is not known whether it is possible to elect a leader in an hypercube in just $O(n)$ messages even when it is not oriented (Problem 9.9).

6 Election in Complete Networks

We have seen how structural properties of the network can be effectively used to overcome the additional difficulty of operating in a fully symmetric graph. For example, in oriented hypercubes, we have been able to achieve $O(n)$ costs, i.e., comparable to those obtainable in trees.

On the other hand, a ring has very few links and no additional structural property capable of overcoming the disadvantages of symmetry. In particular, it is so sparse (i.e., $m = n$) that it has the worst diameter among regular graphs (to reach the furthermost node, a message must traverse $d = n/2$ links) and no short cuts. It is thus no surprising that election requires $\Omega(n \log n)$ messages.

The ring is the sparsest network and it is an extreme in the spectrum of regular networks. At the other end of the spectrum lies the complete graph K_n; in K_n, each node is connected directly to every other node. It is thus the densest network

$$m = \frac{1}{2}n(n-1)$$

and the one with smallest diameter

$$d = 1.$$

Another interesting property is that K_n contains every other network G as a subgraph ! Clearly, physical implementation of such a topology is very expensive.

Let us examine how to exploit such very powerful features to design an efficient election protocol.

6.1 Stages and Territory

To develop an efficient protocol for election in complete networks, we will use *electoral stages* as well as a new technique, *territory acquisition*.

In *territory acquisition*, each *candidate* tries to "capture" its neighbours (i.e., all other nodes) one at a time; it does so by sending a Capture message containing its id as well as the number of nodes captured so far (the *stage*). If the attempt is successful, the attacked neighbour becomes *captured*, the *candidate* enters the next stage and continues; otherwise, the *candidate* becomes *passive*. The *candidate* that is successful in capturing all entities becomes the *leader*.

Summarizing, at any time an entity is either *candidate*, or *captured*, or *passive*. A *captured* entity remembers the id, the stage and the link to its "owner" (i.e., the entity that captured it). Let us now describe an electoral stage.

1. A *candidate* entity x sends a Capture message to a neighbour y.
2. If y is *candidate*, the outcome of the attack depends on the stage and the id of the two entities:
   (a) if $stage(x) > stage(y)$, the attack is successful.
   (b) If $stage(x) = stage(y)$, the attack is successful if $id(x) < id(y)$; otherwise x becomes *passive*.
   (c) If $stage(x) < stage(y)$, x becomes *passive*.
3. If y is *passive*, the attack is successful.
4. If y is already *captured*, then x has to defeat y's owner z before capturing y. Specifically, a Warning message with x's id and stage is send by y to its owner z.
   (a) If z is a *candidate* in a higher stage, or in the same stage but with a smaller id than x, then the attack to y is *not* successful: z will notify y which will notify x.
   (b) In all other cases (z is already *passive* or *captured*, z is a *candidate* in a smaller stage, or in the same stage but with a larger id than x), then the attack to y is *successful*: z notifies x via y, and if *candidate* it becomes *passive*.

5. If the attack is successful, y is *captured* by x, x increments $stage(x)$ and proceeds with its conquest.

Notice that each attempt from a *candidate* costs exactly two messages (one for the Capture, one for the notification) if the neighbour is also a *candidate* or *passive*; instead, if the neighbour was already *captured*, two additional messages will be sent (from the neighbour to its owner, and back).

The strategy just outlined will indeed solve the election problem (Exercise 9.65). Eventhough each attempt costs only four (or fewer) messages, the overall cost can be prohibitive; this is due to the fact that the number n_i of candidates at level i can in general be very large (Exercise 9.66).

To control the number n_i, we need to ensure that a node is captured by at most one candidate in the same level. In other words, the territories of the candidates in stage i must be mutually disjoint. Fortunately, this can be easily achieved.

First of all, we provide some intelligence and decisional power to the *captured* nodes:

- If a *captured* node y receives a Capture message from a *candidate* x that is in a stage smaller than the one known to y, then y will immediately notify x that the attack is unsuccessful.

As a consequence, a *captured* node y will only issue a Warning for an attack at the highest level known to y. A more important change is the following.

- If a *captured* node y sends a Warning to its owner z about an attack from x, y will wait for the answer from z (i.e., locally enqueue any subsequent Capture message in same or higher stage) before issuing another Warning.

As a consequence, if the attack from x was successful (and the stage increased), y will send to the new owner x any subsequent Warning generated by processing the enqueued Capture messages. After this change, the territory of any two candidates in the same level are guaranteed to have no nodes in common (Exercise 9.64).

Protocol *CompleteElect* implementing the strategy we have just designed is shown in Figures 43, 44, and 45.

Let us analyze the cost of the protocol.

How many candidates there can be in stage i ? Since each of them has a territory of size i and these territories are disjoint, there cannot be more than $n_i \leq n/i$ such *candidates*. Each will originate an attack which will cost at most 4 messages; thus, in stage i there will be at most $4n/i$ messages.

Let us now determine the number of stages needed for termination. Consider the following fact: if a *candidate* has conquered a territory of size $n/2+1$, no other *candidate* can become

PROTOCOL CompleteElect.

- $\mathcal{S}$ = {ASLEEP, CANDIDATE,PASSIVE, CAPTURED, FOLLOWER, LEADER}; $\mathcal{S}_{INIT}$ = {ASLEEP}; $\mathcal{S}_{TERM}$ = {FOLLOWER, LEADER}.
- Restrictions: **RI**$\cup\{CompleteGraph\}$.

```
ASLEEP
      Spontaneously
      begin
           stage:= 1; value:= id(x);
           Others:= N(x);
           next ← Others;
           send(‘‘Capture’’, stage, value) to next;
           become CANDIDATE;
      end

      Receiving(‘‘Capture’’, stage*, value*)
      begin
           send(‘‘Accept’’, stage*, value*) to sender;
           stage:= 1;
           owner:= sender;
           ownerstage:= stage* +1;
           become CAPTURED;
      end


CANDIDATE
      Receiving(‘‘Capture’’, stage*, value*)
      begin
           if  (stage* < stage) or ((stage* = stage) and (value* > value)) then
              send(‘‘Reject’’, stage) to sender;
           else
              send(‘‘Accept’’, stage*, value*) to sender;
              owner:= sender;
              ownerstage:= stage* +1;
              become CAPTURED;
           endif
      end

      Receiving(‘‘Accept’’, stage, value)
      begin
           stage:= stage+1;
           if  stage ≥ 1 + n/2 then
              send(‘‘Terminate’’) to N(x);
              become LEADER;
           else
              next ← Others;
              send(‘‘Capture’’, stage, value) to next;
           endif
      end

(CONTINUES ...)
```

Figure 43: Protocol *CompleteElect* (I)

```
CANDIDATE
      Receiving(``Reject'', stage*)
      begin
           become PASSIVE;
      end

      Receiving(``Terminate")
      begin
           become FOLLOWER;
      end

      Receiving(``Warning'', stage*, value*)
      begin
           if  (stage* < stage) or ((stage* = stage) and (value* > value)) then
              send(``No'', stage) to sender;
           else
              send(``Yes", stage*) to sender;
              become PASSIVE;
           endif
      end


PASSIVE
      Receiving(``Capture'', stage*, value*)
      begin
           if  (stage* < stage) or ((stage* = stage) and (value* > value)) then
              send(``Reject'', stage) to sender;
           else
              send(``Accept'', stage*, value*) to sender;
              ownerstage:= stage* +1;
              owner:= sender;
              become CAPTURED;
           endif
      end

      Receiving(``Warning'', stage*, value*)
      begin
           if  (stage* < stage) or ((stage* = stage) and (value* > value)) then
              send(``No'', stage) to sender;
           else
              send(``Yes", stage*) to sender;
           endif
      end

      Receiving(``Terminate")
      begin
           become FOLLOWER;
      end

(CONTINUES ...)
```

Figure 44: Protocol *CompleteElect* (II)

```
CAPTURED
      Receiving(‘‘Capture’’, stage*, value*)
      begin
           if  stage* < ownerstage then
               send(‘‘Reject’’, ownerstage) to sender;
           else
               attack:= sender;
               send(‘‘Warning", value*, stage*) to owner;
               close N(x) − {owner};
           endif
      end

      Receiving(‘‘No", stage*)
      begin
           open N(x);
           send(‘‘Reject’’, stage*) to attack;
      end

      Receiving(‘‘Yes", stage*)
      begin
           ownerstage:= stage*+1;
           owner:= attack;
           open N(x);
           send(‘‘Accept’’, stage*, value*) to attack;
      end

      Receiving(‘‘Warning’’, stage*, value*)
      begin
           if  (stage* < ownerstage) then
               send(‘‘No’’, ownerstage) to sender;
           else
               send(‘‘Yes", stage*) to sender;
           endif
      end

      Receiving(‘‘Terminate")
      begin
           become FOLLOWER;
      end
```

Figure 45: Protocol *CompleteElect* (III)

leader. Hence, a *candidate* can become *leader* as soon as it reaches that stage (it will then broadcast a termination message to all nodes).

Thus the total number of messages, including the $n-1$ for termination notification, will be

$$n+1+\sum_{i=1}^{n/2} 4n_i \;=\; n+1+4n\sum_{i=1}^{n/2}\tfrac{1}{i} \;=\; 4nH_{n/2}+n+1$$

which gives the overall cost

$$\mathbf{M}[CompleteElect] \leq 2.76\; n\log n - 1.76n + 1 \tag{37}$$

Let us now consider the *time* cost of the protocol. It is not difficult to see that, in the worst case, the ideal time of protocol *CompleteElect* is linear (Exercise 9.67):

$$\mathbf{T}[CompleteElect] = O(n) \tag{38}$$

This must be contrasted with the $O(1)$ time cost of the simple strategy of each entity sending its id immediately to all its neighbours, thus receiving everybody's else id, and determining the smallest id. Obviously, the price we would pay for a $O(1)$ time cost is $O(n^2)$ messages.

Appropriately combining the two strategies we can actually construct protocols that offer optimal $O(n\log n)$ message costs with $O(n/\log n)$ time (Exercise 9.68).

The time can be further reduced at the expenses of more messages. In fact, it is possible to design an election protocol that, for any $\log n \leq k \leq n$, uses $O(nk)$ messages and $O(n/k)$ time in the worst case (Exercise 9.69).

6.2 Surprising Limitation

We have just developed an efficient protocol for election in complete networks. Its cost is $O(n\log n)$ messages. Observe that this is the same as we were able to do in *ring* networks (actually the multiplicative constant is *worse*).

Unlike rings, in complete networks each entity has a direct link to all other entities, and there is a total of $O(n^2)$ links. By exploiting all this communication hardware, we should be able to do better than in rings, where there are only n links, and where entities can be $O(n)$ far apart.

The most surprising result about complete networks is that, in spite of having available the largest possible amount of connection links and a direct connect between any two entities, for election they *do not fare better than ring networks.*

In fact, any election protocol will require in the worst case $\Omega(n\log n)$ messages. That is,

Property 6.1 $\mathcal{M}(\mathbf{Elect}/IR; K) = \Omega(n\log n)$

To see why this is true, observe that any election protocol also solves the *wake-up* problem; to become *defeated* or *leader* an entity must have been active (i.e. *awake*). This simple observation has dramatic consequences. In fact, any Wake-up protocol requires at least $.5n \log n$ messages in the worst case (Property **??**); thus, any Election protocol requires in the worst case the same number of messages.

This implies that, as far as election is concerned, the very large expenses due to the physical construction of $m = (n^2 + n)/2$ links are not justifiable, since the same performance and operational costs can be achieved with only $m = n$ links arranged in a ring.

6.3 Harvesting the Communication Power

The lower bound we have just seen carries a very strong and rather surprising message for network development: insofar election is concerned, complete networks are not worth the large communication hardware costs. The facts that Election is a basic problem and its solutions are routinely used by more complex protocols, makes this message even stronger.

The message is surprising because the complete graph, as we mentioned, has the most communication links of any network, and the shortest possible distance between any two entities.

To overcome the limit imposed by the lower bound, and thus to harvest the communication power of complete graphs, we need the presence of some additional tools (i.e., properties, restrictions, etc.). The question becomes: which tool is powerful enough? Since each property we assume restricts the applicability of the solution, our quest for a powerful tool should be focused on the least restrictive ones.

In this section we will see how to answer this question. In the process, we will discover some intriguing relationships between port numbering and consistency, and shed light on some properties of whose existence we already had an inkling in earlier section.

We will fist examine a particular labelling of the ports that will allow us to make full use of the communication power of the complete graph.

The first step consists in viewing a complete graph K_n as a ring R_n, where any two non-neighbouring nodes have been connected by an additional link, called *chord*. Assume that the label associated at x to link (x, y) is equal to the (clockwise) distance from x to y in the ring. Thus, each link in the ring is labelled 1 in the clockwise direction, and $n - 1$ in the other. In general, if $\lambda_x(x, y) = i$ then $\lambda_y(y, x) = n - i$ (see Figure 47); this labelling is called *chordal*.

Let us see how election can be performed in a complete graph with such a labelling.

First of all, observe the following: since the links labelled 1 and $n-1$ form a ring, the entities could ignore all the other links and execute on this subnet an election protocol for rings, e.g., *Stages*. This approach will yield a solution requiring $2n \log n$ messages in the worst case, thus already improving on *CompleteElect*. But we can do better than that.

Consider a *candidate* entity x executing stage i: it will send an election message in both

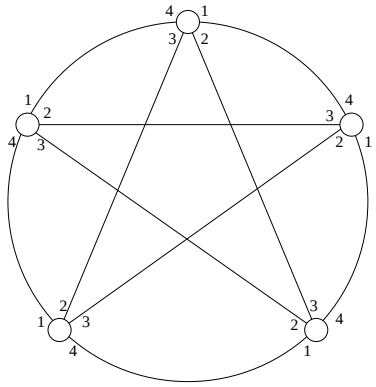


Figure 46: A complete graph with chordal labelling. The links labelled 1 and 4 form a ring.

directions, which will travel along the ring until they reach another *candidate*, say y and z (see Figure **??**). This operation will require the transmission of $d(x, y) + d(x, z)$ messages. Similarly, x will receive the Election messages from both y and z, and decide whether it survives this stage or not, based on the received ids.

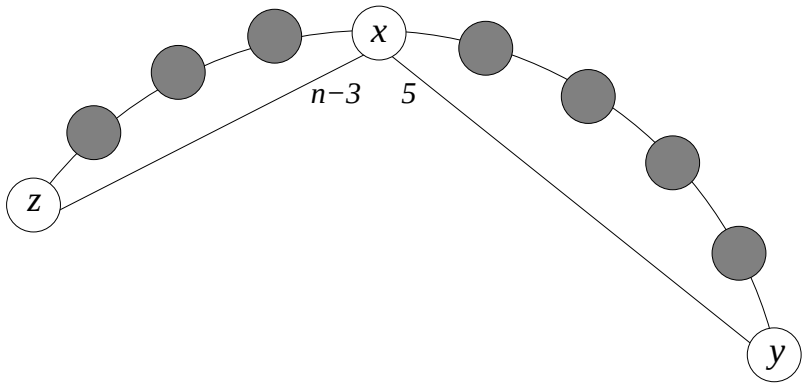


Figure 47: A complete graph with chordal labelling. The links labelled 1 and 4 form a ring.

Now, in a complete graph there exists a direct link between x and y, as well as between x and z; thus, a message from one to the other could be conveyed with only one transmission. Unfortunately, x does not know which of its $n - 1$ links connect it to y or to z; y and z are in a similar situation. In the example of Figure **??**, x does not know that y is the node at distance 5 along the ring (in the clockwise direction), and thus the port connecting x to it is the one with label 5. If it did, those three defeated nodes in between them could be

bypassed. Similarly, x does not know that z is at distance -3 (i.e., at distance 3 in the counterclockwise direction) and thus reachable through port $n-3$.

However, this information can be acquired !

Assume that the Election message contains also a counter, initialized to one, which is increased of one unit by each node forwarding it. Then, a candidate receiving the Election message knows exactly which port label connects it to the originator of that message. In our example, the election message from y will have a counter equal to 5 and will arrive from link 1 (i.e., counterclockwise), while the message from z will have a counter equal to 3 and will arrive from link $n-1$ (i.e., clockwise). From this information x can determine that y can be reached directly though port 5, and z is reachable through link $n-3$. Similarly, y (resp. z) will know that the direct link to x is the one labelled $n-5$ and (resp. 3).

This means that, *in the next stage*, these chords can be used instead of the corresponding segments of the ring, thus saving message transmissions. The net effect will be that, in stage $i+1$, the candidates will use the (smaller) ring composed only of the chords determined in the previous stage; that is, messages will be sent only on the links connecting the candidates of stage i, thus completely bypassing all entities defeated in stage $i-1$ or earlier.

Assume in our example that x enters stage $i+1$ (and thus both y and z are defeated); it will prepare an election message for the candidates in both directions, say u and v, and will send it directly to y and to z. As before, x does not know where u and v are (i.e., which of its links connect it to them) but, as before, it can determine it.

The only difference is that the counter must be initialized to the *weight* of the chord: thus the counter of the Election message sent by x directly to y is equal to 5, and the one to z is equal to 3. Similarly, when an entity forwards the Election message through a link, it will add to the counter the weight of that link.

Summarizing, in each stage the candidates will execute the protocol in a smaller ring. Let $R(i)$ be the ring used in stage i; initially $R(1) = R_n$. Using the ring protocol *Stages* in each stage, the number of messages we will be transmitting will be exactly $2(n(1) + n(2) + \ldots + n(k))$, where $n(i)$ is the size of $R(i)$ and $k \leq \log n$ is the number of stages; an additional $n-1$ messages will be used for the leader to notify termination.

Observe that all the rings $R(2), \ldots, R(k)$ do not have links in common (Exercise 9.70). This means that if we consider the graph G composed of all these rings, then the number of links $m(G)$ of G is exactly $m(G) = n(2) + \ldots + n(k)$. Thus to determine the cost of the protocol, we need to find out the value of $m(G)$.

This can be determined in many ways. In particular, it follows from a very interesting property of those rings. In fact, each $R(i)$ is "contained" in the interior of $R(i+1)$: all the links of $R(i)$ are chords of $R(i+1)$, and these chords do not cross. This means that the graph G formed by all these rings is *planar*; that is can be drawn in the plane without any edge crossing. A well known fact of planar graphs is that they are sparse, i.e., contain very few links: no more than $3(n-2)$ (if you did not know it, now you do). This means that our graph G has $m(G) \leq 3n-6$. Since our protocol, which we shall call *Kelect*, uses $2(n(1) + m(G)) + n$ messages in the worst case, and $n(1) = n$, we have

$$\mathbf{M}[\text{Kelect}] < 8n - 12$$

A less interesting, but more accurate measurement of the message costs follows from observing that the nodes in each ring $R(i)$ are precisely the entities that were candidates in stage $i-1$; thus, $n(i) = n_{i-1}$. Recalling that $n_i \leq \frac{1}{2}n_{i-1}$, and since $n_1 = n$, we have $n(1) + n(2) + \ldots + n(k) \leq n + \sum_{i=1}^{k-1} n_i < 3n$, which will give

$$\mathbf{M}[Kelect] < 7n \tag{39}$$

The conclusion is that the chordal labeling allows us to finally harvest the communication power of complete graphs, and do better than in ring networks.

7 Election in Chordal Rings ⋆

We have seen how election requires $\Omega(n \log n)$ messages in rings, and can be done with just $O(n)$ messages in complete networks provided with chordal labelling. Interestingly, oriented rings and complete networks with chordal labelling are part of the same family of networks, known as *loop networks* or *chordal rings*.

A chordal ring $C_n\langle d_1, d_2, ..., d_k\rangle$ of size n and k-chord structure $\langle d_1, d_2, ..., d_k\rangle$, with $d_1 = 1$, is a ring R_n of n nodes $\{p_0, p_1, ..., p_{n-1}\}$, where each node is also directly connected to the nodes at distance d_i and $N - d_i$ by additional links called *chords*. The link connecting two nodes is labelled by the distance which separates these two nodes on the ring, i.e. following the order of the nodes on the ring: node p_i is connected to the node $p_{i+d_j \bmod n}$ through its link labelled d_j (as shown in Figure 48). In particular, if the link between p and q is labelled d at p, this link is labelled $n - d$ at q.

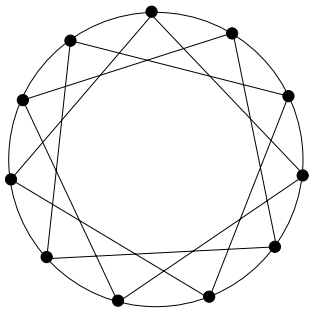


Figure 48: Chordal ring $C_{11}\langle 1, 3\rangle$

Note that the oriented ring is the chordal ring $C_n\langle 1\rangle$ where label 1 corresponds to "right", and $n-1$ to "left". The complete graph with chordal labelling is the chordal ring $C_n\langle 1, 2, 3, \cdots, \lfloor n/2 \rfloor\rangle$ In fact, rings and complete graphs are two extreme topologies among chordal rings.

Clearly, we can exploit the techniques we designed for complete graph with chordal labelling to develop an efficient election protocol for the entire class of chordal ring networks. The strategy is simple:
1. Execute an efficient ring election protocol (e.g., *Stages* or *Alternate*) on the outer ring. As we did in *Kelect*, the message sent in a stage will carry a counter, updated using the link labels, that will be used to compute the distance between two successive *candidates*.
2. Use the chords to bypass *defeated* nodes in the next stage.

Clearly, the more distances can be "bypassed" by chords, the more messages we will be able to save. As an example, consider the chordal ring $C_n\langle 1, 2, 3, 4..., t\rangle$, where every entity is connected to its distance-t neighbourhood in the ring. In this case (Exercise 9.76), a leader can be elected with a number of messages no more than

$$O(n + \tfrac{n}{t} \log \tfrac{n}{t})$$

A special case of this class is the complete graphs, where $t = \lfloor n/2 \rceil \rfloor$; in it we can bypass any distance in a single "hop" and, as we know, the cost becomes $O(n)$.

Interestingly, we can achieve the same $O(n)$ result with fewer chords. In fact, consider the chordal ring $C_n\langle 1, 2, 4, 8..., 2^{\lceil \log n/2 \rceil}\rangle$; it is called *double cube* and $k = \lceil \log n \rceil$. In a double cube, this strategy allows election with just $O(n)$ messages (Exercise 9.78), like if we were in a complete graph and had all the links !

At this point, an interesting and important question is what is the smallest sets of links that must be added to the ring in order to achieve a linear election algorithm. The double cube indicates that $k = O(\log n)$ suffice. Surprisingly, this can be significantly further reduced (Problem 9.12); furthermore (Problem 9.13), the $O(n)$ cost can be obtained even if the links have arbitrary labels !

Lower bounds

The class of chordal rings is quite large; it includes rings and complete graphs, and the cost of electing a leader varies greatly depending on the structure. For example we have already seen that the complexity is $\Theta(n \log n)$ and $\Theta(n)$ in those two extreme chordal rings.

We can actually establish precisely the complexity of the election problem for the entire class of chordal rings $C_n^t = C_n\langle 1, 2, 3, 4..., t\rangle$. In fact, we have (Exercise 9.77):

$$\mathcal{M}(\textbf{Elect}/IR; C_n^t) = \Omega(n + \frac{n}{t} \log \frac{n}{t}) \tag{40}$$

Notice that this class includes the two extremes. In view of the matching upper bound (Exercise 9.76), we have

Property 7.1 *The message complexity of* **Elect** *in* C_n^t *under* **IR** *is* $\Theta(n + \frac{n}{t} \log \frac{n}{t})$

8 Universal Election Protocols

We have so far studied in detail the election problem in *specific* topologies; i.e., we have developed solution protocols for restricted classes of networks, exploiting in their design all the graph properties of those networks so to minimize the costs and increase the efficiency of the protocols. In this process we have learned some strategies and principles which are however very general (e.g., the notion of *electoral stages*), as well as the use of known techniques (e.g., broadcasting) as modules of our solution.

We will now focus on the main issue, the design of *universal election protocols*, that is protocols which run in every network, requiring no a priori knowledge of the topology of the network nor of its properties (not even its size !). In terms of communication software, such protocols are obviously are totally *portable*, and thus highly desirable.

We will describe two such protocols, radically different from each other. The first, *Mega-Merger*, constructs a rooted spanning-tree, is highly efficient (optimal in the worst case); the protocol is however rather complex both in terms of specifications and analysis, and its correctness is still without a simple formal proof. The other, *Yo-Yo*, is a minimum-finding protocol which is exceedingly simple to specify and to prove correct; its real cost is however not yet known.

8.1 Mega-Merger

In this section we will discuss the design of an efficient algorithm for leader election, called *Mega-Merger*. This protocol is topology independent (i.e., universal) and constructs a (minimum cost) rooted spanning tree of the network.

Nodes are small villages, and edges are roads each with a different name and distance. The goal is to have all villages merge into one large megacity. A city (even a small village will be considered such) always tries to merge with the closest neighbouring city.

When merging, there are several important issues that must be resolved. First and foremost, the *naming of the new city*. The resolution of this issue depends on how far the involved cities have progressed in the merging process, i.e. on the *level* they have reached, and on whether the merger decision is shared by both cities.

The second issue to be resolved during a merging is the decision of which roads of the new city will be serviced by *public transports*. When a merge occurs, the roads of the new city serviced by public transports will be the roads of the two cities already serviced plus only the shortest road connecting them.

Let us clarify some of these concepts and notions, as well as the basic rules of the game.

- (1) A *city* is a rooted tree; the nodes are called *districts*, the root is also known as *downtown*.
- (2) Each city has a level and a unique name; all districts eventually know the name and the level of their city.

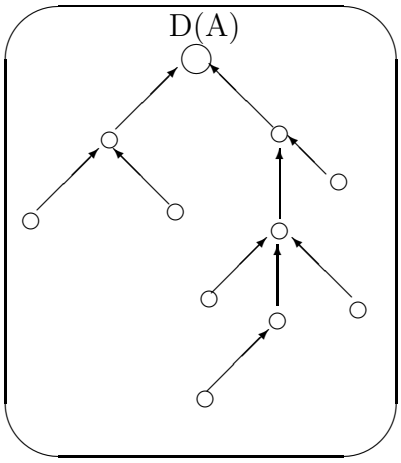


Figure 49: A city is a tree rooted in its downtown.

- (3) Edges are *roads*, each with a distinct name and distance. The city roads are only those serviced by public transport.
- (4) Initially, each node is a city with just one district, itself, and no roads. All cities are initially at the same level.

Note that, as a consequence of rule (1), every district knows the direction (i.e., which of its links in the tree leads) to its downtown (Figure 49).

- (5) A city must merge with its closest neighbouring city. To request the merging, a *Let-us-Merge* message is sent on the shortest road connecting it to that city.
- (6) The decision to request for a merger must originate from downtown, and there cannot be more than one request at a time from the same city.
- (7) When a merge occurs, the roads of the new city serviced by public transports will be the roads of the two cities already serviced plus only the shortest road connecting them.

Thus, to merge, the downtown of city A will first determine the shortest link, which we shall call the *merge link*, connecting it to a neighbouring city; once this is done, a *Let-us-Merge* is sent through that link; the message will contain information identifying the city and the chosen merge link. Once the message reaches the other city, the actual merger can start to take place. Let us examine the components of this entire process in some details.

We will consider city A, denote by D(A) its downtown, by $level(A)$ its current level, and by $e(A) = (a, b)$ the *merge link* connecting A to its closest neighbouring city; let B be such a city.

8.1.1 Processing the merger request

Once the *Let-us-Merge* message from a in A reaches the district b of B, three cases are possible.

If the two cities have the same level and each asks to merge with the other, we have what is called a *friendly merger*: the two cities merge into a new one that, to avoid any conflict, will be named after the shortest road connecting them; the level of the new megacity will be one unit more than that of the two cities forming it. In short,

- (8) If $level(A) = level(B)$ and the merge link chosen by A is the same as that chosen by B (i.e., $e(A) = e(B)$), then A and B perform a *friendly merger*.

If a city asks a merger with a city of *higher* level, it will just be *absorbed*; that is, it will acquire the name and the level of the other city:

- (9) If $level(A) < level(B)$, A is *absorbed* in B.

In all other cases, the request for merging and thus the decision on the name are *postponed* ! In particular, if a city asks to merge with a city of *smaller* level, the request will *wait* until that city reaches an equal or larger level than its own. (Why? the second city, like every city, is in the process of merging with some other city and, as a result, will eventually increase its level.) Similarly, if the two cities have the same level but the merger was requested by only one of them, that one will *wait* until the other city reaches a larger level than its own, and it will then be absorbed. That is,

- (10) If $level(A) = level(B)$ but the merge link chosen by A is not the same as that chosen by B (i.e., $e(A) \neq e(B)$),then the merge process of A with B is *suspended* until the level of b's city is larger than that of A.
- (11) If $level(A) > level(B)$, the merge process of A with B is *suspended*: x will locally enqueue the message until the level of b's city is at least as large as the one of A.

Let us see these rules in more details, and examine what happens when the *Let-us-Merge* message sent from node a in city A arrives to a node b in city B. Node b will be called the *entry point* of the request from A to B, and node a the *exit point*. Let us denote by d_A and d_B the two downtowns. Assume for the moment that no concurrent merging requests are sent to B from other cities. The entry point b will perform the processing of the request. Let $level(B)$ and $name(B)$ be the values known to b. (Note that, unknown to b, these values might be changing.) The possible outcomes will be *absorption*, *friendly merger* or *suspension*.

Absorption

The absorption process is the conclusion of a merger request sent by A to a city with a higher level (rule 9). As a result, city A becomes part of city B acquiring the name, the downtown,

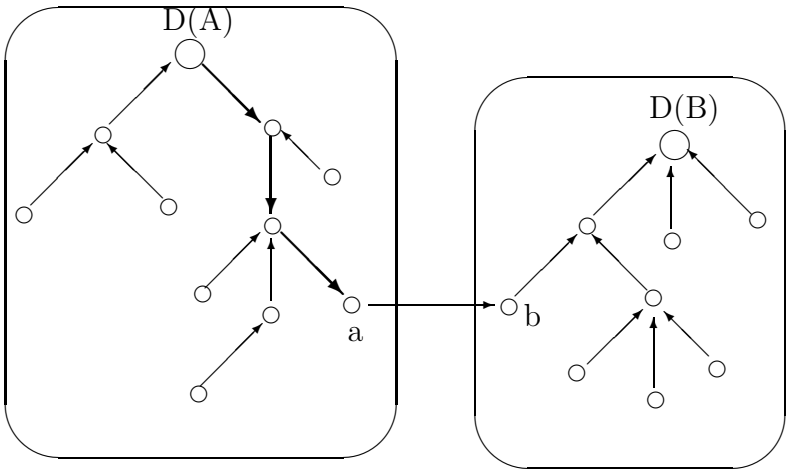


Figure 50: Absorption. The logical direction of the links (in bold) from the downtown to the exit point of A is "flipped".

---

and the level of B. This means that, during absorption:
i) the logical orientation of the roads in A must be modified so that they are directed towards the new downtown (so rule (1) is satisfied);
ii) all districts of A must notified of the name and level of the city they just joined (so rule (2) is satisfied).
All these requirements can be easily and efficiently achieved.
First of all, the entry point b will notify a (the exit point of A) that the outcome of the request is absorption, and it will include in the message all the relevant information about B (name and level). Once a receives this information, it will broadcast it in A; as a result, all districts of A will join the new city, and know its name and its level.
To transform A so it is rooted in the new downtown is fortunately simple. In fact, it is sufficient to logically direct towards B the link connecting a to b, and to "flip" the logical direction only of the edges in the path from the exit point a to the old downtown of A (Exercise 9.79). The latter can be done as follows: each of the districts of B on the path from a to d_A, when it receives the broadcast from a, it will locally direct towards B two links: the one from which the broadcast message is received and the one towards its old downtown.

Friendly merger
If A and B are at the same level in the merging process (i.e., $level(A) = level(B)$) and want to merge with each other (i.e., $e(A) = e(B)$), we have a friendly merger. Notice that, if this is the case, also a must receive a *Let-us-Merge* message from b. The two cities now become one with a new downtown, a new name, and an increased level. The new downtown will be the one of a and b that has smaller id (recall that we are working under the **ID** restriction). The name of the new city will be the name of the road connecting a and b (recall that, by rule (3), we are assuming unique names and lengths for the roads). The level will be increased by one unit. Both a and b will independently compute the new name, level, and downtown.

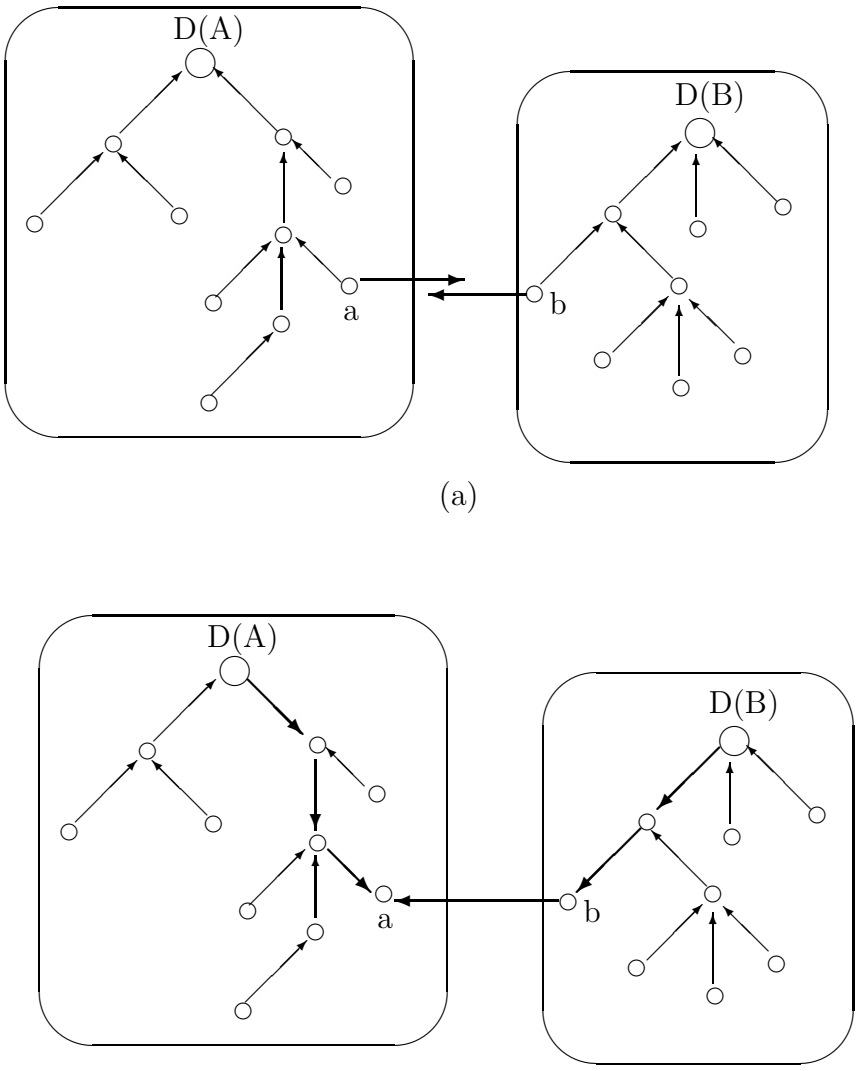


Figure 51: Friendly Merger (a) The two cities have the same level and choose the same merge link. (b) The new downtown is the exit node (a or b) with smallest id.

Then each will broadcast this information to its old city; as a result, all districts of A and B will join the new city, and know its name and its level.
Both A and B must be transformed so they are rooted in the new downtown. As discussed in the case of absorption, it is sufficient to "flip" the logical direction only of the edges in the path from the a to the old downtown of A, and of those in the path from b to the old downtown of B (Figure 51).

Suspension
In two cases (rules (10) and (11)), the merge request of A must be suspended: b will then locally enqueue the message until the level of its city is such that it can apply rule (8) or (9). Notice that, in case of suspension, nobody from city A knows that their request has been suspended; because of rule (6), no other request can be launched from A.

8.1.2 Choosing the merging edge

According to rule (6), the choice of the merging edge $e(A)$ in A is made by the downtown $D(A)$; according to rule (5), $e(A)$ must be the shortest road connecting A to a neighbouring city. Thus, $D(A)$ needs to find the minimum length among all the edges incident on the nodes of the rooted tree A; this will be done as following.

- (5.1) Each district a_i of A determines the length d_i of the shortest road connecting it to another city (if none goes to another city, then $d_i = \infty$).
  (5.2) $D(A)$ computes the smallest of all the d_i.

Part (5.2) is easy to accomplish; it is just a minimum-finding in a rooted tree, which we have discussed in Section **??**.

Concentrate on part (5.1), and consider a district a_i; it must find, among its incident edges the shortest one that leads to an another city.

IMPORTANT. Obviously, a_i does not need to consider the *internal roads* (i.e., those that connect it to other districts of A). Unfortunately, if a link is *unused* i.e., no message has been sent or received through it, it is impossible for a_i to know if this road is internal or leads to a neighbouring city (Figure 52). In other words, a_i must try also the internal unused roads.

Thus, a_i will determine the shortest unused edge e, prepare a *Outside?* message, send it on e and wait for a reply. Consider now the district c on the other side of e, which receives this message; c knows the $name(C)$ and the $level(C)$ of its city (which could however be changing).

If $name(A) = name(C)$ (recall that the message contains the name of A), c will reply *Internal* to a_i, the road e will be marked as internal (and no longer used in the protocol) by both districts, and a_i will restart its process.

If $name(A) \neq name(C)$ it does not necessarily mean that the road is not internal. In fact, it is possible that, while c is processing this message, its city C is being absorbed by A ! Observe that, in this case, $level(C)$ must be *smaller* than $level(A)$ (because, by rule (8) only a city with smaller level will be absorbed). This means that, if $name(A) \neq name(C)$ but

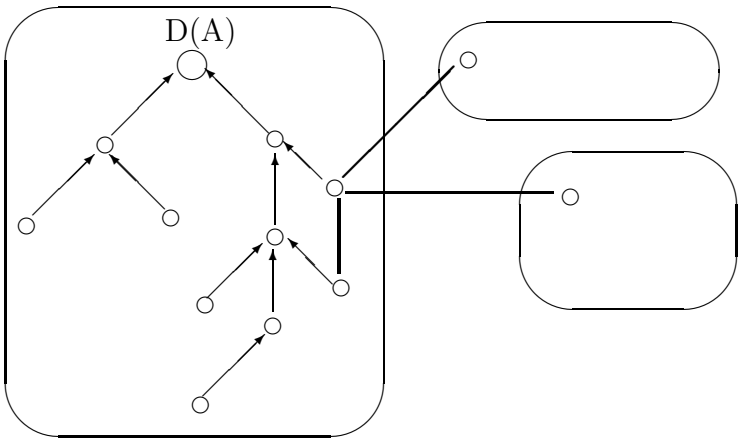


Figure 52: Some unused links (in bold) might lead back to the city.

$level(C) \geq level(A)$, then C is not being absorbed by A, and it is for sure a different city; thus c will reply *External* to a_i, which will have thus determined what it was looking for: $d_i = length(e)$.

The only case left is when $name(A) \neq name(C)$ and $level(C) < level(A)$, in which c cannot give an answer. So, it will not: c will postpone the reply until the level of its city becomes greater or equal to that of A. Note that this means that the computation in A is suspended until c is ready !

Concluding: to determine if a link is internal should be simple but, due to concurrency, the process is not trivial nor obvious.

8.1.3 Detailed Cases

We have just seen in details the process of determining the merge link as well as the rules governing a merger. There is still few issues which require a more detailed treatment.

Discovering a friendly merger
We have seen that, when the *Let-us-Merge* message from A to B arrives at b, if $level(A) = level(B)$, the outcome will be different (friendly merger or postponement) depending on whether $e(A) = e(B)$ or not. Thus, to decide if it is a friendly merger, b needs to know both $e(A)$ and $e(B)$. When the *Let-us-Merge* message sent from a arrives to b, b knows $e(A) = (a, b)$.
QUESTION: how does b know $e(B)$? The answer is interesting.
As we have seen, the choice of $e(B)$ is made by the downtown $D(B)$ which will forward the merger request message of B towards the exit point.
If $e(A) = e(B)$, b is the exit point and thus it will *eventually* receive the message to be sent to a; then (and only then) b will know the answer to the question, and that it is dealing with a friendly merger !
If $e(A) \neq e(B)$, b is not the exit point. Note that, unless b is on the way from downtown

$D(B)$ to the exit point, b will *not* even know what $e(B)$ is !
Thus, what really happens when the *Let-us-Merge* message from A arrives at b, is the following. If b has received already a *Let-us-Merge* message from its downtown to be sent to a, then b knows that is a friendly merger; also a will know when it receives the request from b. (NOTE FOR HACKERS: thus, in this case, no reply to the request is really necessary.) Otherwise b does not know; thus it waits: if it is a friendly merger, sooner or later the message from its downtown will arrive and b will know; if B is requesting another city, eventually the level of b's city will increase becoming greater than $level(A)$ (which, since A is still waiting for the reply, cannot increase), and thus result in A being absorbed.

Interaction between absorption and link calculation
Because of the asynchronous nature of the system and its unpredictable (though, finite) communication delays, it will likely be the case that different cities and districts will be at different levels at the same time. In fact, our rules take explicitly into account the interaction between neighbouring cities at different levels. There are a few situations where the application of the rules will not be evident. A situation that requires attention is due to the interaction between merge link calculation and absorption. Consider the *Let-us-Merge* message sent by a on merge link $e(A) = (a, b)$ to b, which is computing the merge-link for its city B; from the message, b finds out that $level(A) < level(B)$, and thus A will have to be absorbed in B. What should b do ? On one hand, b could start the absorption process of A first and then continue its computation of the merge-link; in this way, the districts of A will also be involved in the computation of the merge link for B (expanding the list of possibilities). On the other hand, b could pretend that the message from a has still not arrived yet, conclude its computation of the merge-link, and only then start the absorption process of A; in this way, it will possibly save the messages of involving A in this calculation of the merge edge of B. There are also other options in which b makes a more reasoned decision based on the length of $e(A)$ (Exercise 9.80).

8.1.4 The Effects of Postponements and Concurrency

A city only carries out one merger requests at the time, but it can be asked concurrently by several cities, which in turn can be asked by several others. Some of these requests will be postponed (because the level is not right, or the entry node does not (yet) know what the answer is, etc.) Due to communication delays, some districts will be taking decisions based on information (level and name of its city) which is obsolete. It is not difficult to imagine very intricate and complex scenarios which can easily occur.

How do we know that, in spite of concurrency and postponements and communication delays, everything will eventually work out ? How can we be assured that some decisions will not be postponed forever, i.e. there will not be *deadlock* ? What guarantees that, in the end, the protocol terminates and a single leader will be elected ? In other words, how do we know that the protocol is correct ?

Due to its complexity and the variety of scenarios that can be created, there is no satisfactory complete proof of the correctness of the *Mega-Merger* protocol. We will discuss here a partial proof which will be sufficient for our learning purposes.

Progress and Deadlock

We will first discuss the progress of the computation and the absence of deadlock . To do so, let us pinpoint the cases when the activity of a city C is halted by a district d of another city D. This can occur only when computing the merge edge, or when requesting a merger on the merge edge $e(C)$; more precisely, there are four cases:
(i) when computing the merge edge, a district c of C sends the *Outside?* message to d and D has a smaller level than C;
(ii) after receiving the *Let-us-Merge* message on the merge edge $e(C) = (c, d)$, d realizes that D has smaller level than C;
(iii) after receiving the *Let-us-Merge* message on the merge edge $e(C) = (c, d)$, d realizes that D and C have the same level but it is not a friendly merger.
(iv) after receiving the *Let-us-Merge* message on the merge edge $e(C) = (c, d)$, d realizes that D and C have the same level but it does not know that it is a friendly merger.

In cases (i)-(iii) the activities of C are suspended and will be resolved (if the protocol is correct) only in the "future", i.e. after D changes level. Case (iv) is different from the others, as it will be resolved within the "present" (i.e., in this level); it is in fact a *delay* rather than a *suspension*.

Observe that, if there is no suspension, there is no problem.

Property 8.1 *If a city at level l will* not *be suspended, its level will eventually increase, unless it is the megacity.*

To see why this is true, consider the operations performed by a city C at a level l: compute the merge edge, and send a merge request on the merge edge. If it is not suspended, its merge request arrives at a city D with either a larger level (in which case, C is absorbed and its level becomes $level(D)$), or the same level and same merge-edge (in which case, the two cities friendly merge and their level increases).

So, only suspensions can create problems. But not necessarily so:

Property 8.2 *Let city C at level l be suspended by a district d in city D. If the level of the city of D becomes greater than l, C will no longer be suspended and its level will increase.*

This is because, once the level of D becomes greater than the level of C, d can answer the *Outside?* message in case (i), as well as the *Let-us-Merge* message in cases (ii) and (iii).

Thus, the only real problem is the presence of a city suspended by another whose level will not grow. We are going now to see that this can not occur.

Consider the *smallest* level l of any city at time t, and concentrate on the cities $\mathcal{C}$ operating at that level at that time.

Property 8.3 *No city in $\mathcal{C}$ will be suspended by a city at higher level.*

This is because, for a suspension to exist, the level of D can *not* be greater that the level of C (see the cases above).

Thus, if a city $C \in \mathcal{C}$ is suspended, it is for some other city $C' \in \mathcal{C}$. If C' will not be suspended at level l, its level will increase; when that happens, C will no longer be suspended. In other words, there will be no problems as long as there are no cycles of suspensions within $\mathcal{C}$; that is, as long as there is no cycle $C_0, C_1, \ldots, C_{k-1}$ of cities of $\mathcal{C}$ where C_i is suspended by C_{i+1} (and the operation on the indices are modulo k). The crucial property is the following:

Property 8.4 *There will be no cycles of suspensions within* $\mathcal{C}$*.*

The proof of this property is based heavily on the fact that each edge has a unique length (we have assumed that !), and that the merge edge $e(C)$ chosen by C is the shortest of all the unused links incident on C. Remember this fact, and let us proceed with the proof.

By contradiction, assume that the property is false. That is, assume there is a cycle $C_0, C_1, \ldots, C_{k-1}$ of cities of $\mathcal{C}$ where C_i is suspended by C_{i+1} (the operation on the indices are modulo k). First of all observe that since all these cities are at the same level, then the reason they are suspended can only be that each is involved in an "unfriendly" merger, i.e., case (iii). Let us examine the situation more closely: each C_i has chosen a merge edge $e(C_i)$ connecting it to C_{i+1}; thus C_i is suspending C_{i-1} and is suspended by C_{i+1}. Clearly both $e(C_{i-1})$ and $e(C_i)$ are incident on C_i. By definition of merging edge (recall what we said at the beginning of the proof), $e(C_i)$ is shorter than $e(C_{i-1})$ (otherwise C_i would have chosen it instead); in other words, the length d_i of the road $e(C_i)$ is smaller than the length d_{i11} of $e(C_{i+1})$. This means that $d_0 > d_1 > \ldots > d_{k-1}$; but since it is a circle of suspensions, C_{k-1} is suspended by C_0, that is $d_{k-1} > d_0$. We have reached a contradiction, which implies that our assumption that the property does not hold is actually false; thus the property is true.

As a consequence of the property, all cities in $\mathcal{C}$ will eventually increase their level: first the ones involved in a friendly merger, next those that had chosen them for a merger (and thus absorbed by them), then those suspended by the latter, and so on.

This implies that, *at no time there will be deadlock* and *there is always progress*: use the properties to show that the ones with smallest level will increase their value; when this happens, again the ones with smallest level will increase it, and so on. That is

Property 8.5 *Protocol* Mega-Merger *is deadlock-free and ensures progress.*

Termination
We have just seen that there will be no deadlock, and that progress is guaranteed. This means that the cities will keep on merging, and eventually the megacity will be formed. The problem is how to detect that this has happened. Recall that no node has knowledge of the network, not even of its size (it is not part of the standard set of assumptions for election); how does an entity finds out that all the nodes are now part of the same city ? Clearly it is sufficient for just one entity to determine termination (as it can then broadcast it to all the others).

Fortunately, *termination detection* is simple to achieve; as one might have suspected, it is the downtown of the megacity that will determine that the process is terminated.

Consider the downtown $D(A)$ of city A, and the operations it performs: it coordinates the computation of the merge link, and then originates a merge request to be sent on that link. Now, the merge link is the shortest road going to *another* city. If A is already the megacity, there are no other cities; hence all the unused links are internal. This means that, when computing the merge link, every district will explore every unused link left and discover that each one of them is internal; it will thus choose ∞ as its length (meaning that it does not have any outgoing links). This means that the minimum-finding process will return ∞ as the smallest length. When this happens, $D(A)$ understands that the mega-merger is completed, and can notify all others. (Notification is not really necessary: Exercise 9.82).

Since the megacity is a rooted tree with the downtown as its root, $D(A)$ becomes the *leader*; in other words

Property 8.6 *Protocol* Mega-Merger *correctly elects a leader.*

8.1.5 Cost

In spite of the complexity of protocol *Mega-Merger*, the analysis of its cost is not overly difficult. We will first determine how many levels there can be, and then calculate the total number of messages transmitted by entities at a given level.

The number of levels
A district acquires a larger level because its city has been either absorbed or involved in a friendly merger. Notice that, when there is absorption, only the districts in one of the two cities increase their level, and thus the max level in the system will not be increased. The max level can only increase after a friendly merger.

How high can the max level be ? We can find out by linking the minimum number of districts in a city to the level of the city.

Property 8.7 *A city of level i has at least 2^i districts.*

This can be proved easily by induction. It is trivially true at the beginning (i.e., $i = 0$). Let it be true for $0 \leq i \leq k-1$. A level k city can only be created by a friendly merger of two level $k-1$ cities; hence, by inductive hypothesis, such a city will have at least $2\ 2^{k-1} = 2^k$ districts; thus the property is true also for $i = k$.

As a consequence

Property 8.8 *No city will reach a level greater than* $\log n$.

The number of messages per level
There are all types of activities performed to increase the level. Consider a level i; some districts will reach this level from level $i-1$, while others directly from a even lower level.

We want to count how many messages are exchanged in this transition. Definitely we need to count all the messages required by the friendly merger of cities of level $i-1$, including the cost of finding the common merge link. We also need to count all the messages incurred by absorptions to level i by cities of lower levels, including the cost of finding their merge link. In the calculation of the merge link we will *not* include, for the moment, the messages sent to internal roads; they will be counted separately later.

Consider the cost of all the friendly mergers creating cities of level i. In each of the level $i-1$ cities involved in a friendly merger, three activities are performed: the computation of the merge link; the forwarding the message from downtown to that merge link; the broadcast from the exit point of the information about the new city (and the directive to compute the new merge link). Using the protocol for minimum-finding in rooted trees, the computation of the merge link costs 2 $n(A)$ messages, in a city A with $n(A)$ districts. The forwarding of the *Let-us-Merge* message from the downtown $D(A)$ to the merge edge $e(A) = (a, b)$ will cause at most $n(A)$ transmissions. Similarly, the broadcast from a of the information about the new city costs only $n(A) - 1$ messages (recall we are in a tree). In other words, for a level $i-1$ city A involved in a friendly merger, the cost will be at most $4n(A) - 1$.

Consider now a city C which it is absorbed into another of level i. Also in C the same three activities are performed. The computation of the merge link and the forwarding the message from downtown to that merge link will cost 2 $n(C)$ and at most $n(C)$ messages, respectively. The broadcast in C of the new level and name (and possibly the directive to compute the new merge link) is started by the entry point thus requires one message more than in the case of a friendly merger, for a total of $n(C)$ messages. In other words, for a city C absorbed into one of level $i-1$, the cost will be at most $4n(C)$.

This means that the total cost $Cost(i)$ for level i will be

$$Cost(i) = \sum_{A \in Merge(i)} (4n(A) - 1) + \sum_{C \in Absorb(i)} 4n(C)$$

where $Merge(i)$ and $Absorb(i)$ are the set of the cities becoming of level i through friendly merger and absorption, respectively. Since all these cities are disjoint,

$$\sum_{B \in (Merge(i) \ \cup Absorb(i)} \ n(B) \leq n$$

hence

Property 8.9 $Cost(i) \leq 4n$

The number of useless messages

In the calculation so far we have excluded the messages sent, during the search for a merge link, to unused links that turn out to be internal roads. These messages are in a sense "useless" since they do not bring to a merger; but they are also unavoidable. Let us measure their number. On any such road there will be two messages, the *Outside?* message and the *Internal* reply. So, we only need to determine the number of such roads. These roads are not part of the city (i.e., not serviced by public transport). Since the final city is a tree, the

total number or the publicly serviced roads is exactly $n-1$. Thus the total number of the other roads is exactly $m-(n-1)$. This means that the total number of useless messages will be

Property 8.10 $Useless = 2(m-n+1)$

The total
Combining Properties 8.8, 8.9, and 8.10, we obtain the total number of messages exchanged in total by protocol *Mega-Merger* during all its levels of execution. To these, we need to add the $n-1$ messages due to the downtown of the megacity broadcasting termination (eventhough these could be saved: Exercise 9.82), for a total of

$$\mathbf{M}[Mega-Merger] \;\leq\; 2m + 4n\log n + 1 \tag{41}$$

8.1.6 Road Lengths and Minimum-Cost Spanning Trees

In all the previous discussions we have made some non-standard assumptions about the edges. We have in fact assumed that each link has a value, which we called length, and that those values are unique. (This last assumption provides for free unique names to the links, as the unique length can be used as the link's name.)

The existence of link values is not uncommon. In fact, dealing with networks, usually there is a value associated with a link denoting e.g., the cost of using that link, the transmission delays incurred when sending a message through it, etc.

In these situations, when constructing a spanning-tree (e.g. to use for broadcasting), the prime concern is how to construct the one of *minimum cost*, i.e., where the sum of the values of its link is as small as possible. For example, if the value of the link is the cost of using it, a minimum-cost spanning tree is one where broadcasting would be cheapest (regardless of who is the originator of the broadcast). Not surprisingly, the problem of constructing a minimum-cost spanning tree is important and heavily investigated.

We have seen that protocol *Mega-Merger* constructs a rooted spanning tree of the network. What we are going to see now is that this tree is actually the unique minimum-cost spanning-tree of the network. We are also going see how the non-standard assumptions we have made about the existence of unique lengths can be easily removed.

Minimum-Cost Spanning Trees
In general, a network can have several minimum cost spanning trees. For example, if all links have the same value (or have no value), then every spanning tree is minimal. On the other hand,

Property 8.11 *If the link values are distinct, a network has a unique minimum-cost spanning tree.*

Assuming that there are distinct values associated to the links, protocol *Mega-Merger* constructs a rooted spanning tree of the network. What we are going to see now is that this tree is actually the unique minimum-cost spanning-tree of the network.

To see why this is the case, we must observe a basic property of the minimum-cost spanning tree T. A *fragment* of T is a subtree of T.

Property 8.12 *Let A be a fragment of T, and let e be the link of minimum value among those connecting A to other fragments; let B be the fragment connected by A. Then the tree composed by merging A and B through e is also a fragment of T.*

This is exactly what the *Mega-Merger* protocol does: it constructs the minimum-cost spanning tree T (the megacity) by merging fragments (cities) through the appropriate edges (merge link). Initially each node is a city and, by definition, a single node is a fragment. In general, each city A is a fragment of T; its merge link is chosen as the shortest (i.e., minimum-value) link connecting A to any neighbouring city (i.e., fragment); hence, by Property 8.12, the result of the merger is also a fragment.

Notice that the correctness of the process depends crucially on Property 8.11, and thus on the distinctness of the link values.

Creating Unique Lengths

We will now remove the assumptions that there are values associated to the links, and these values are unique.

If there are no values (the more general setting), then a unique value can be easily given to each link using the fact that the nodes have unique ids: to link $e = (a, b)$ associate the sorted pair $d(e) =< Min\{id(a), id(b)\}, Max\{id(a), id(b)\} >$, and use the lexicographic ordering to determine which edge has smaller length. So, for example, the link between nodes with ids 17 and 5 will have length $< 5, 17 >$, which is smaller than $< 6, 5 >$ but greater than $< 4, 32 >$. To do this requires however that each node knows the id of all its neighbours. This information can be acquired in a pre-processing phase, in which every node sends to its neighbours its id (and will receive theirs from them); the cost will be two additional messages on each link. Thus, even if there are no values associated to the links, it is possible to use protocol *Mega-Merger*. The price we have to pay is $2m$ additional messages.

If there are values but they are not (known to be) unique, they can be made so, again using the fact that the nodes have unique ids. To link $e = (a, b)$ with value $v(e)$ associate the sorted triple $d(e) =< v(e), Min\{id(a), id(b)\}, Max\{id(a), id(b)\} >$. Thus, links with the same values will now be associated to different lengths. So, for example, the link between nodes with ids 17 and 5 and value 7 will have length $< 7, 5, 17 >$, which is smaller than $< 7, 6, 5 >$ but greater than $< 7, 4, 32 >$. Also in this case, each node needs to know the id of all its neighbours. The same pre-processing phase will achieve the goal with only $2m$ additional messages.

Summarizing, protocol *Mega-Merger* is a universal protocol which constructs a (minimum-cost) spanning tree and returns it rooted in a node, thus electing a leader. If there are

no initial distinct values on the links, a pre-processing phase need to be added, in which each entity exchanges its unique id with its neighbours; then the actual execution of the protocol can start. The total cost of the protocol (with or without pre-processing phase) is $O(m + n \log n)$ which, we will see, is worst case optimal.

The main drawback of *Mega-Merger* is its design complexity, which makes any actual implementation difficult to verify.

8.2 YO-YO

We will now examine another universal protocol for leader election. Unlike the previous one, it has simple specifications, and its correctness is simple to establish. This protocol, called *YO-YO*, is a Minimum-Finding algorithm and consists of two parts: a pre-processing phase, and a sequence of iterations. Let us examine them in detail.

8.2.1 Setup

In the pre-processing phase, called *Setup*, every entity x exchanges its id with its neighbours. As a result, it will receive the id of all its neighbours. Then x will logically orient each incident link (x, y) in the direction of the entity (x or y, with the largest id. So, if $id(x) = 5$ and its neighbour y has $id(y) = 7$, x will orient (x, y) towards y; notice that also y will do the same. In fact, the orientation of each link will be consistent at both end nodes.

Consider now the directed graph $\vec{G}$ so obtained. There is a very simple but important property:

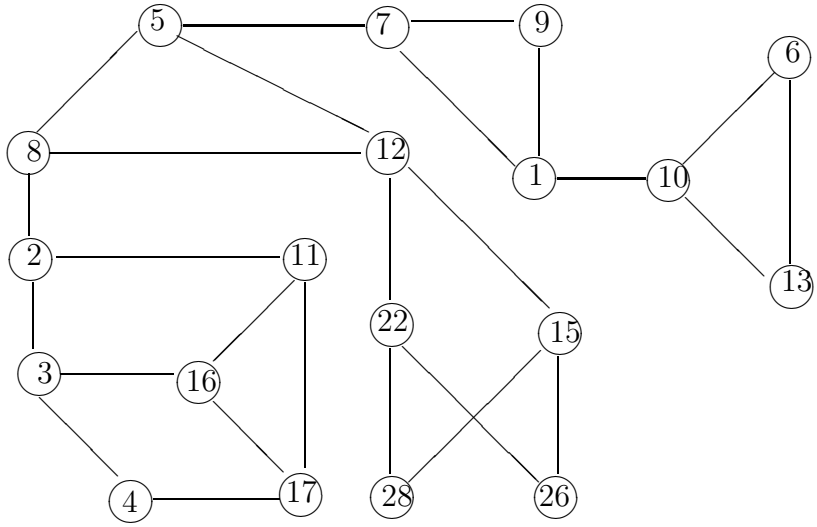
Property 8.13 *$\vec{G}$ is acyclic.*

To see why this is true, consider by contradiction the existence of a directed cycle $x_0, x_1, \ldots, x_k$; this means that $id(x_0) < id(x_1) < \ldots < id(x_{k-1})$ but, since it is a cycle, $id(x_{k-1}) < id(x_0)$, which is impossible.

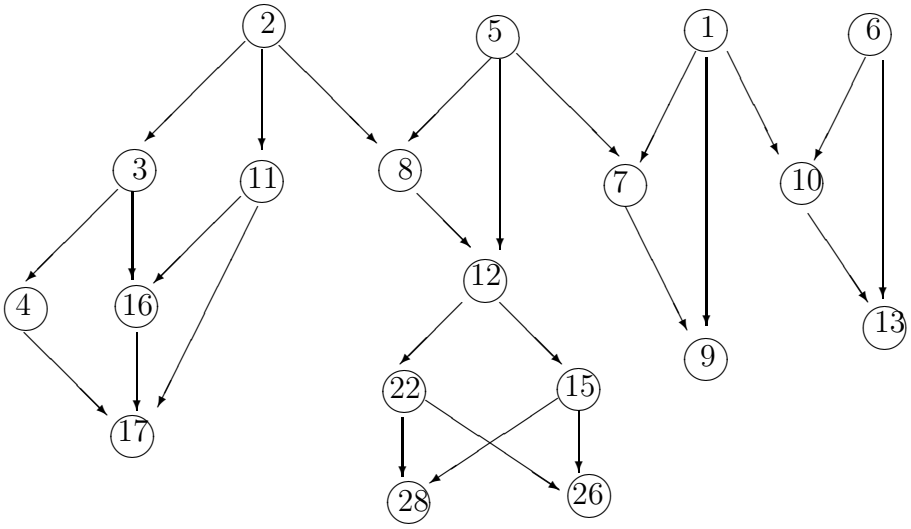
This means that $\vec{G}$ is a *directed acyclic graph* (*DAG*). In a DAG there are three types of nodes:
- *source* is a node where all the links are out-edges; thus, a source in $\vec{G}$ is a node with an id smaller than that of all its neighbours, i.e., it is a *local minimum*;
- *sink* is a node where all the links are in-edges; thus, a sink in $\vec{G}$ is a node whose id is larger than that of all its neighbours, i.e., it is a *local maximum*;
- *internal node* is a node which is neither a source nor a sink.

As a result of the setup, each node will know whether it is a source, a sink, or an internal node. We will also use the terminology of "down" referring to the direction towards the sinks, and "up" referring to the direction towards the sources (See Figure 53.

Once this preprocessing is completed, the second part of the algorithm is started.



(a)



(b)

Figure 53: In the Setup phase, (a) the entities know their neighbours' ids, and (b) orient each incident link towards the smaller id, creating a DAG.

8.2.2 Iteration

The core of the protocol is a sequence of iterations. Each iteration acts as an electoral stage in which some of the candidates are removed from consideration. As *Yo-Yo* is a minimum-finding protocol, only the local minima (i.e., the sources) will be *candidates* (Figure 54.

Each iteration is composed of two parts, or *phases*, called *Yo-* and *-Yo*.

YO-
This phase is started by the sources. Its purpose is to propagate to each sink the smallest among the values of the sources connected[3] to that sink (see Figures 54(a) and 56(a)).

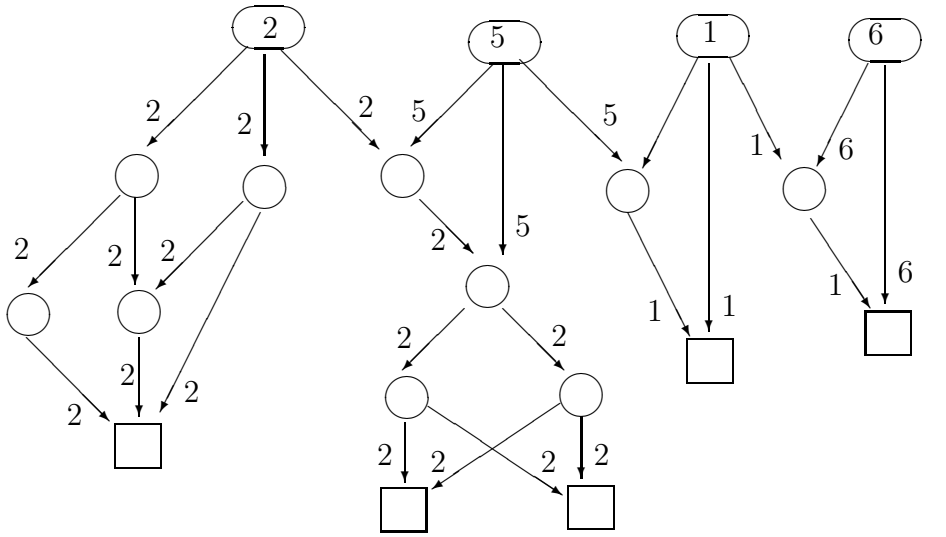
- (1) A source sends its value down to all its out-neighbours.
- (2) An internal node waits until it receives a value from all its in-neighbours. It then computes the minimum of all received values and sends it down to its out-neighbours.
- (3) A sink waits until it receives a value from all its in-neighbours. It then computes the minimum of all received values, and starts the second part of the iteration.

-YO
This phase is started by the sinks. Its purpose is to eliminate some candidates, transforming some sources into sinks or internal nodes. This is done by having the sinks informing their connected sources of whether their id is the smallest seen so far (see Figure 54(b)).

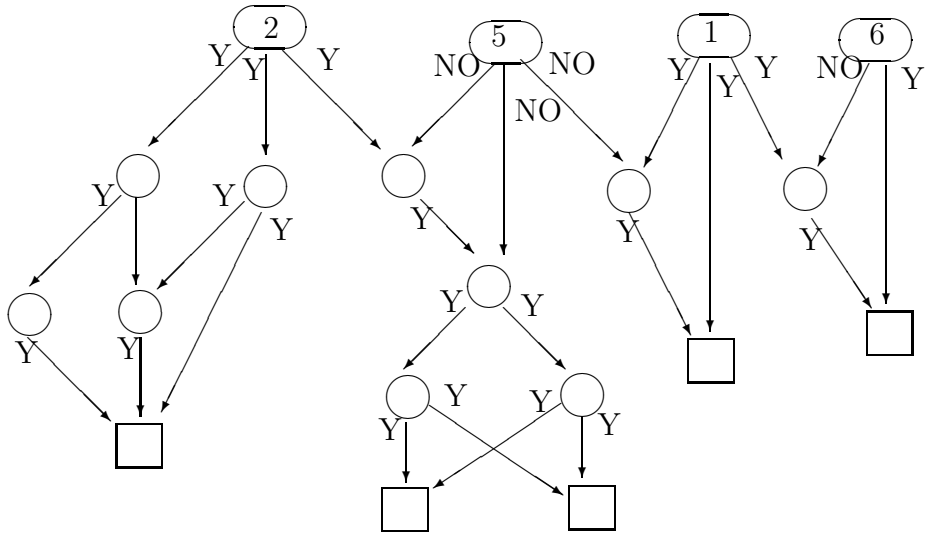
- (4) A sink sends *YES* to all in-neighbours from which the smallest value has been received. It sends *NO* to all the others.
- (5) An internal node waits until it receives a vote from all its out-neighbours. If all votes are *YES*, it sends *YES* to all in-neighbours from which the smallest value has been received, and *NO* to all the others. If at least a vote was *NO*, it sends *NO* to all its in-neighbours.
- (6) A source waits until it receives a vote from all its out-neighbours. If all votes are *YES*, it survives this iteration and starts the next one. If at least a vote was *NO*, it is no longer a candidate.

Before the next iteration can be started, the directions on the links in the DAG must be modified, so that only the sources that are still candidate (i.e., those that received only *YES*) will still be sources; clearly the modification must be done without creating cycles. In other words, we must transform the DAG into a new one, whose only sources are the undefeated ones in this iteration. This modification is fortunately simple to achieve. We need only to "flip" the direction of each link where a *NO* vote is sent (see Figures 55(a) and 56(b)). Thus we have two meta-rules for the *-YO* part:

[3] in the sense there is a directed path from the source to that sink;

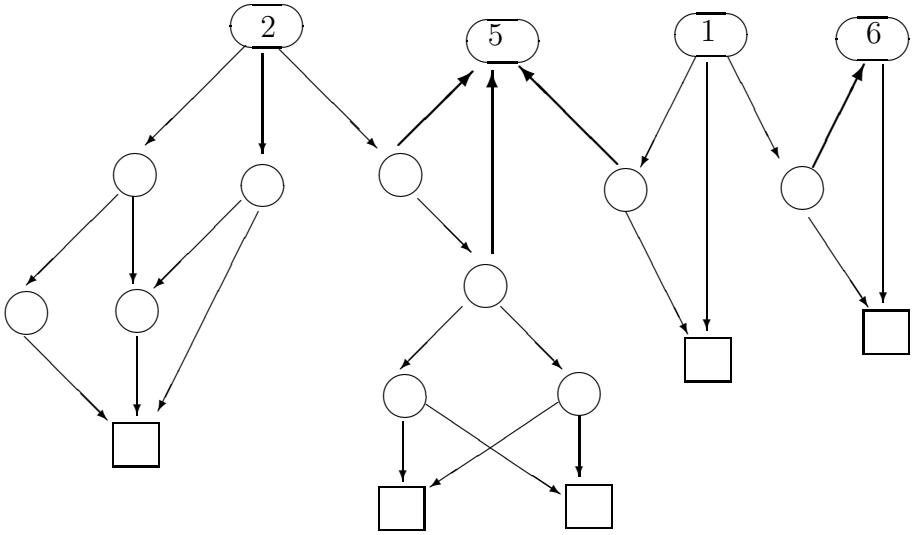


(a)

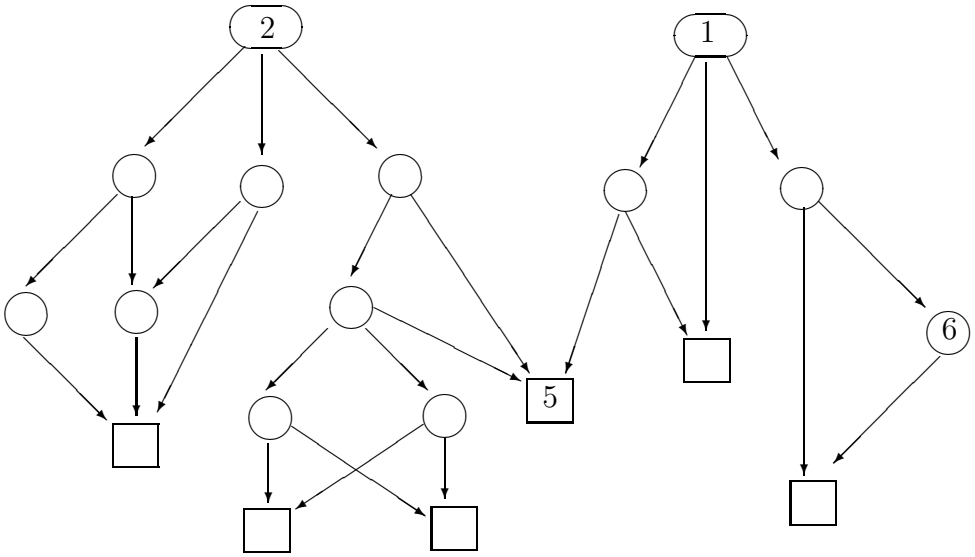


(b)

Figure 54: In the Iteration stage, only the *candidates* are sources. (a) In the Yo- phase, the ids are filtered down to the sinks. (b) In the -YO phase, the votes percolate up to the sources.

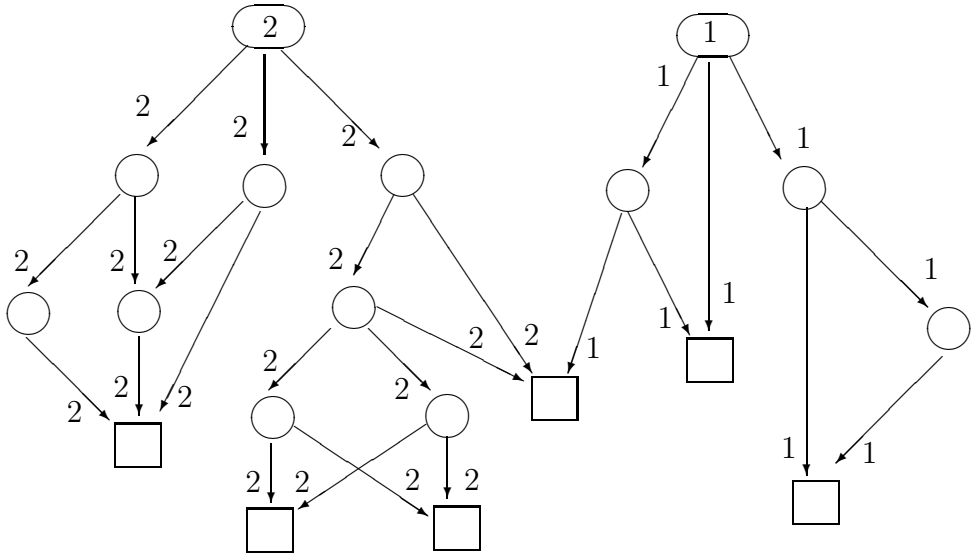


(a)

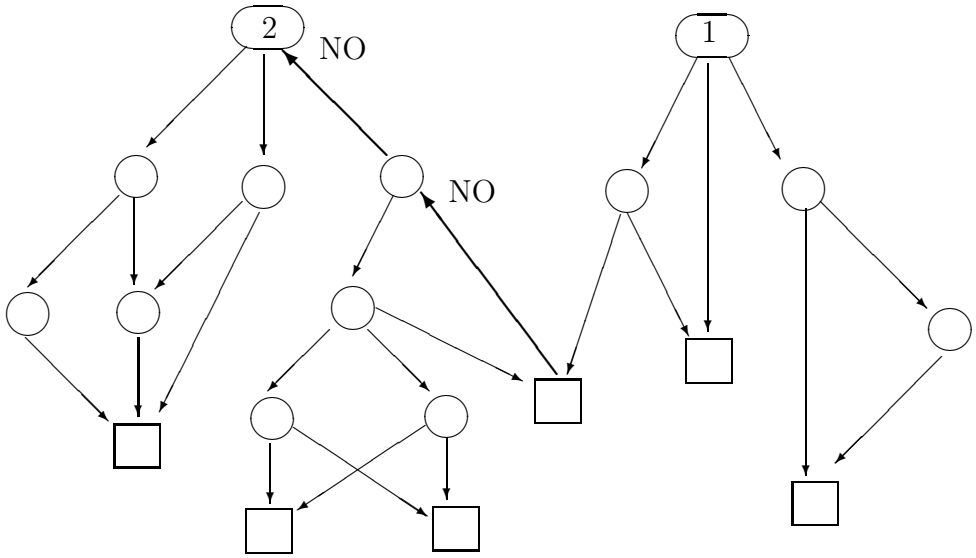


(b)

Figure 55: (a) We flip the logical direction of of the links on which a NO was sent. (b) A new DAG is created, where only the surviving *candidates* are sources.



(a)



(b)

Figure 56: (a) In the Yo- phase, the ids are filtered down to the sinks. (b) In the -Yo phase, we flip the logical direction of of the links on which a NO is sent, creating a new DAG is created, where only the surviving *candidates* will be sources.

- (7) When a node x sends *NO* to an in-neighbour y, it will reverse the (logical) direction of that link (Thus, y becomes now an out-neighbour of x).
- (8) When a node y receives *NO* from an out-neighbour x, it will reverse the (logical) direction of that link (Thus, x becomes now an in-neighbour of y).

As a result, any source which receives a *NO* will cease to be a source; it can actually become a sink ! Some sinks may cease to be such and become internal nodes, and some internal nodes might become sinks. However, no sink or internal node will ever become a source (Exercise 9.84). A new DAG is thus created, where the sources are only those that received all *YES* in this iteration. See Figure 55(b).

Once a node has completed its part in the -*YO* phase, it will know whether it is a source, a sink, or an internal node in the new DAG. The next iteration could now start, initiated by the sources of the new DAG.

Property 8.14 *Applying an iteration to a DAG with more than one source will result into a DAG with fewer sources. The source with smallest value will still be a source.*

In each iteration, some sources (at least one) will be no longer sources; on the other hand the source with smallest value will eventually be the only one left under consideration. In other words, eventually the DAG will have a single source (the overall minimum, say c), and all other nodes are either sinks or internal nodes. How can c determine that it is the only source left, and thus it should become the leader ?

If we were to perform an iteration now, only c's value will be sent in the *YO*- phase, and only YES votes will be sent in the -*YO* phase. The source c will receive only YES votes; but c has received only YES votes in every iteration it has performed (that's why it survived as a source). How can c distinguish that this time is different, that the process should end ? Clearly, we need some additional mechanisms during the iterations.

We are going to add some meta-rules, called *Pruning*, which will allow to reduce the number of messages sent during the iterations, as well as to ensure that termination is detected when only one source is left.

Pruning

The purpose of pruning is to remove from the computation nodes and links that are "useless", do not have any impact on the result of the iteration; in other words, if they where not there, still the same result would be obtained: the same sources would stay sources, and the others defeated. Once a link or a node is declared "useless", during the next iterations it will be considered *non-existent* and thus not used.

Pruning is achieved through two meta-rules.

The first meta-rule is a structural one. To explain it, recall that the function of the sinks is to reduce the number of sources by voting on the received values. Consider now a sink that

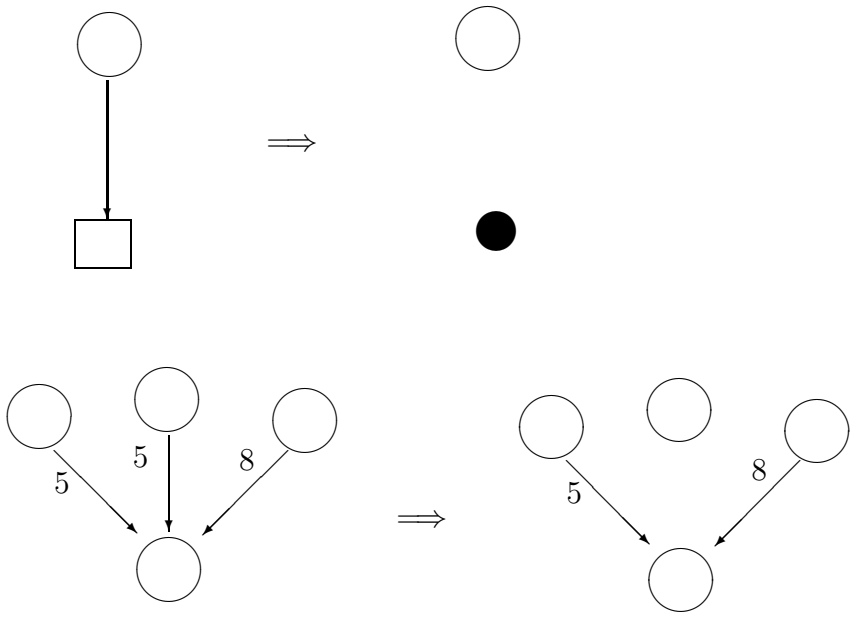


Figure 57: Rules of pruning.

is a leaf (i.e., it has only one in-neighbour); such a node will receive only one value, thus it can only vote *YES*. In other words, a sink leaf can only agree with the choice (i.e., the decision) made by its parent (i.e., its only neighbour). Thus, a sink leaf is "useless".

- (7) If a sink is a leaf (i.e., it has only one in-neighbour), then it is useless; it then asks its parent to be pruned. If a node is asked to prune an out-neighbour, it will do so by declaring useless (i.e., removing from consideration in the next iterations) the connecting link.

Notice that, after pruning a link, a node might become a sink; if it is also a leaf, then it becomes useless !

The other meta-rule is geared towards reducing the communication of redundant information. During *YO-* phase, a (internal or sink) node might receive the value of the same source from more than one in-neighbours; this information is clearly redundant since, to do its job (choose the minimum received value), is enough for the node to receive just one copy of that value. Let x receive the value of source s from in-neighbours $x_1, \ldots, x_k$, $k > 1$. This means that, in the DAG, there are directed paths from s to (at least) k distinct in-neighbours of x. This also means that, if the link between x and one of them, say x_1, did not exist, the value from s would still arrive to x from those other neighbours, $x_2, \ldots, x_k$. In fact, if we had removed the links between x and all those in-neighbours *except one*, x would still have received the value of s from that neighbour. In other words, the links between x and $x_1, \ldots, x_k$ are redundant: it is sufficient to keep one; all others are useless, and can be pruned. Notice that the choice of which of those links should be kept is irrelevant.

- (8) If, in the *YO-* phase, a node receives the same value from more than one in-neighbour, it will ask all of them *except one* to prune the link connecting them, and it will declare those links useless. If a node receives such a request, it will declare useless (i.e., remove from consideration in the next iterations) the connecting link.

Notice that, after pruning a link because of rule (8), a sink might become a leaf, and thus useless (by rule (7)). See Figures 58 and 59.

The pruning rules require communication: in rule (7), a sink leaf need to ask its only neighbour to declare the link between them useless; in rule (8), a node receiving redundant information need to ask some of its neighbours to prune connecting the link. We will have this communication take place during the *-Yo* phase: the message containing the vote will also include the request, if any, to declare that link useless. In other words,

pruning is performed when voting.

Let us return now on our concern on how to detect termination. As we will see, the pruning operations, integrated in the *-Yo* phase, will do the trick. To understand how and why, consider the effect of performing a full iteration (with pruning) on a DAG with only one source.

Property 8.15 *If the DAG has a single source, then, after an iteration, the new DAG is composed of only one node, the source.*

In other words, when there is a single source c, all other nodes will be removed, and c will be the only useful node left ! This situation will be discovered by c when, because of pruning, it will have no neighbours (Figure 60).

8.2.3 Costs

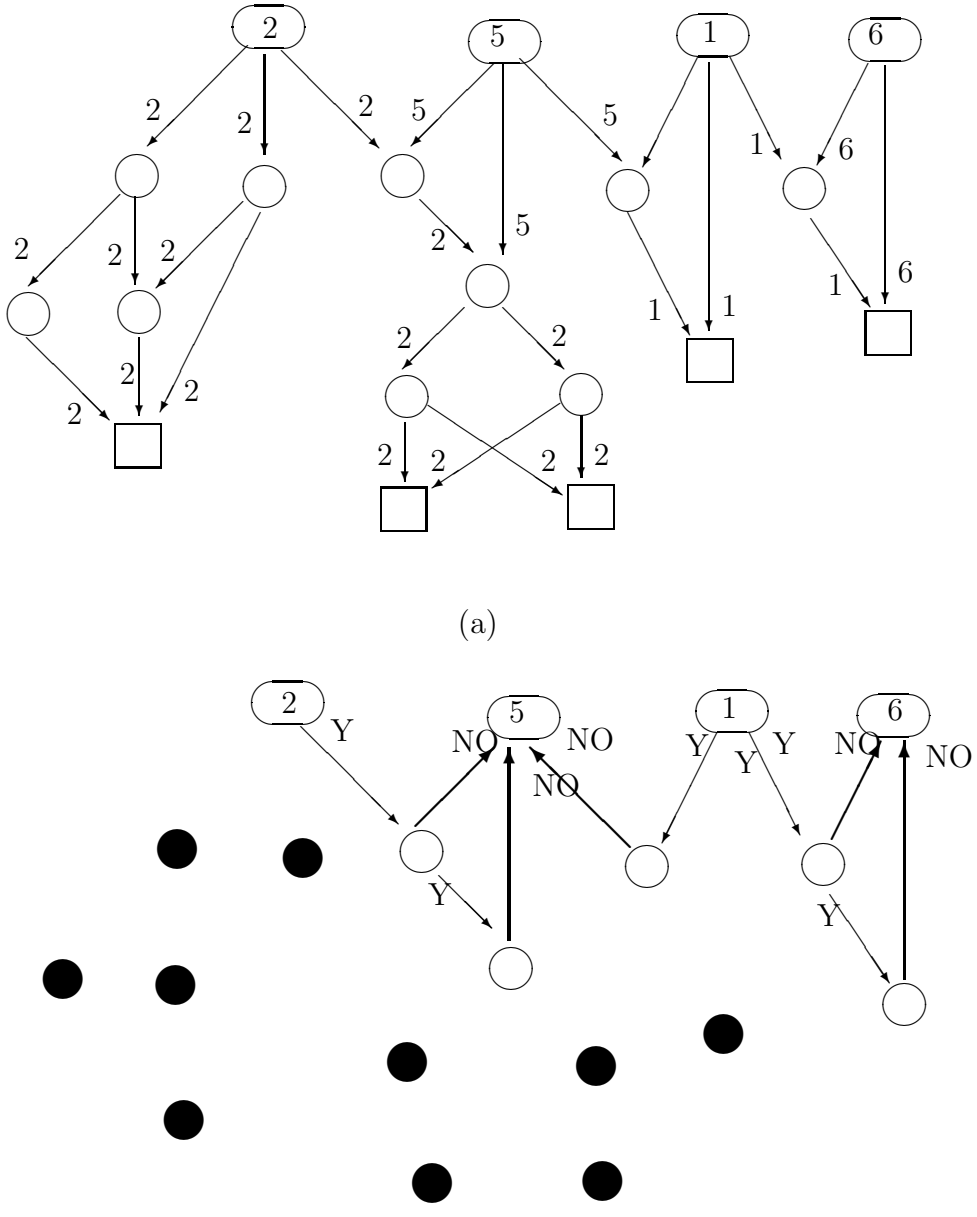
The general formula expressing the costs of protocol *Yo-Yo* is easy to establish; however, the exact determination of the costs expressed by the formula is still an open research problem. Let us derive the general formula.

In the Set-up phase, every node sends its value to all its neighbours; hence, on each link there will be two messages sent, for a total of $2m$ messages.

Consider now an Iteration. In the *Yo-* stage, every useful node (except the sinks) sends a message to its out-neighbours; hence, on each link still under consideration, there will be exactly one message sent. Similarly, in the *-Yo–* stage, every useful node (except the sources) sends a message to its in-neighbours; hence, on each link there will be again only one message sent. Thus, in total in iteration i there will be exactly $2m_i$ messages, where m_i is the number of links in the DAG used at stage i.

The notification of termination from the leader can be performed by broadcasting on the constructed spanning tree with only $n-1$ messages.

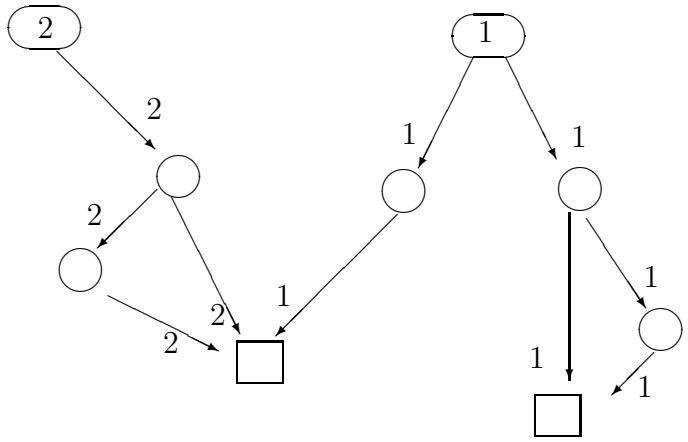
Hence, the total cost will be



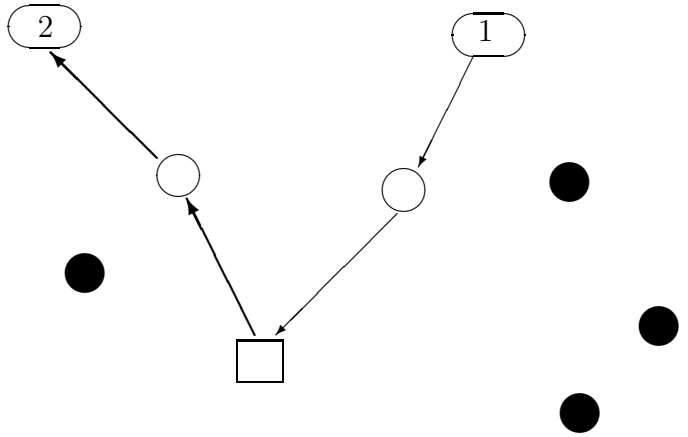
(a)

(b)

Figure 58: The effects of pruning in the first iteration: some nodes (in black) are removed from consideration.



(a)



(b)

Figure 59: The effects of pruning in the second iteration: other nodes (in black) are removed from consideration.

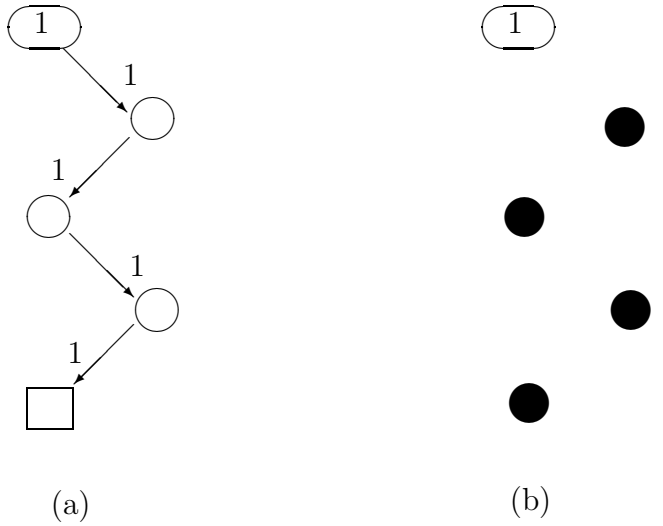


Figure 60: The effects of pruning in the third iteration: termination is detected as the source has no more neighbours in the DAG.

$$2\sum_{i=0}^{k(G)} m_i \ + \ n-1$$

where $m_0 = m$ and $k(G)$ is the total number of iterations on network G.

We need now to establish the number of iterations $k(G)$. Let $D(1) = \vec{G}$ be the original DAG obtained from G as a result of Set-up. Let $G(1)$ be the undirected graph defined as follows: there is a node for each source in $D(1)$, and there is a link between two nodes if and only if the two corresponding sources have a sink in common[4]. Consider now the diameter $d(G(1))$ of this graph.

Property 8.16 *The number of iteration is at most* $\lceil \log \ diam(G(1)) \rceil + 1$.

To see why this is the case, consider any two neighbours a and b in $G(1)$. Since, by definition, the corresponding sources in $D(1)$ have a common sink, at least one of these two sources will be defeated (because the sink will vote YES to only one of them). This means that, if we take any path in $G(1)$, at least half of the nodes on that path will correspond to sources that will cease to be such at the end of this iteration.

Furthermore, if (the source corresponding to) a survives, it will now have a sink in common with each of the undefeated (sources corresponding to) neighbours of b. This means that if we consider the new DAG $D(2)$, the corresponding graph $G(2)$ is exactly the graph obtained by removing the nodes associated to the defeated sources, and linking together the nodes previously at length two. In other words, $d(G(2)) \leq \lceil \ d(G(1))/2 \rceil$.

Similar will be the relationship between the graphs $G(i-1)$ and $G(i)$ corresponding to the DAG $D(i-1)$ of iteration $i-1$ and to the resulting new DAG $D(i)$, respectively. In other

[4]In a DAG, two sources a and b are said to have a common sink c if c is reachable from both a and b.

words, $d(G(i)) \leq \lceil\, d(G(i-1))/2 \rceil$. Observe that $diam(G(i)) = 1$ corresponds to a situation where all sources except one will be defeated in this iteration, and $d(G(i)) = 0$ corresponds to the situation where there is only one source left (which does not know it yet). Since $d(G(i)) \leq 1$ after at most $\lceil \log\, diam(G(1)) \rceil$ iterations, the property follows.

Since the diameter of a graph cannot be greater than the number of its nodes, and since the nodes of $G(1)$ correspond to the sources of $\vec{G}$, we have that

$$k(G) \leq \lceil \log s(\vec{G}) \rceil \leq \lceil \log n \rceil$$

We can thus establish that, *without pruning*, i.e. with $m_i = m$, we have a $O(m \log n)$ total cost:

$$\mathbf{M}[Yo - Yo\ (withoutpruning)] \ \leq\ 2\ m\ \log n + \ l.o.t. \tag{42}$$

The unsolved problem is the determination of the real cost of the algorithm, when the effects of pruning are taken into account.

8.3 Lower Bounds and Equivalences

We have seen a complex but rather efficient protocol, *MegaMerger*, for electing a leader in an arbitrary network. In fact, it uses $O(m + n \log n)$ messages in the worst case. This means that, in a ring network it uses $O(n \log n)$ messages and it is thus optimal, without even knowing that the network is a ring !

The next question we should ask is how efficient a universal election protocol can be. In other words,

what is the complexity of the election problem ?

The answer is not difficult to derive.

First of all observe that any election protocol requires to send a message on *every* link. To see why this is true, assume by contradiction that indeed there is a correct universal election protocol A that, in every network G and in every execution under **IR** does not send a message on every link of G. Consider a network G and an execution of A in G; let z be the entity that becomes *leader* and let $e = (x, y) \in E$ be a link where no message is transmitted by A (Figure 61(a)).

We will now construct a new graph H as follows: we make two copies of G and remove from both of them the edge e; we then connect these two graphs G' and G'' by adding two new edges $e_1 = (x', x'')$ and $e_2 = (y', y'')$, where x' and x'' (resp. y' and y'') are the copies of x (resp. y) in G' and G'' respectively, and where the labels are: $\lambda_{x'}(e_1) = \lambda_{x''}(e_1) = \lambda_x(e)$ and $\lambda_{y'}(e_1) = \lambda_{y''}(e_2) = \lambda_y(e)$. See Figure 61(b).

Run exactly the same execution of A we did in G on the two components G' and G'' of H: since no message was sent along (x, y) in G, this is possible. But since no message was sent

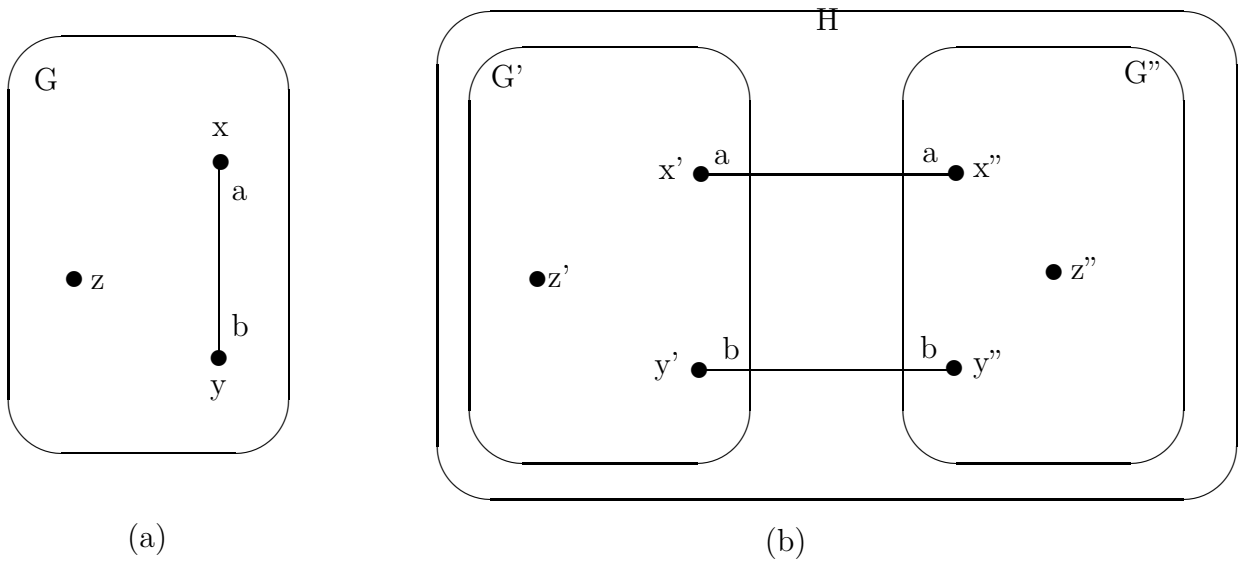


Figure 61: Every universal election protocol must send messages on every link.

along (x, y) in the original execution, x' and x'' will never send messages to each other in the current execution; similarly, y' and y'' will never send messages to each other. This means that the entities of G' will never communicate with the entities of G'' during this execution; thus, they will not be aware of their existence and will operate solely within G'; similarly for the entities of G''.

This means that, when the execution of A in G' terminates, entity z' will become *leader*; but similarly, entity z'' in G'' will become *leader* as well. In other words, two *leaders* will be elected, contradicting the correctness of protocol A. In other words,

$$\mathcal{M}(\textbf{Elect/ IR}) \geq m.$$

This lower bound is powerful enough to provide us with interesting and useful information; for example, it states that $\Omega(n^2)$ messages are needed in a complete graph if you do not know that is a complete graph. On the other hand, we know that there are networks where election requires way more than m messages; for example, in rings $m = n$ but we need $\Omega(n \log n)$ messages. Since a universal election protocol must run in every network, including rings, we can say that in the worst case

$$\mathcal{M}(\textbf{Elect/IR}) \geq \Omega(m + n \log n). \tag{43}$$

This means that protocol *MegaMerger* is worst case *optimal* and we know the complexity of the election problem.

Property 8.17 *The message complexity of election under* **IR** *is* $\Theta(m + n \log n)$

We are now going to see that constructing a spanning-tree **SPT** and electing a leader **Elect** are *strictly equivalent*: any solution to one of them can be easily modified so to solve the other with the same message cost (in order of magnitude).

First of all, observe that , similarly to the Election problem, also **SPT** requires a message to be sent on every link (Exercise 9.86):

$$\mathcal{M}(\mathbf{SPT/IR}) \geq m. \tag{44}$$

We are now going to see how we can construct a spanning-tree construction algorithm from any existing election protocol. Let A be an election protocol; consider now the following protocol B:
(1) Elect a *leader* using A;
(1) The *leader* starts the execution of protocol *Shout*.

Recall that protocol *Shout* (seen in Section **??**) will correctly construct a spanning-tree if there is a unique initiator. Since the *leader* elected in step (1) is unique, a spanning-tree will be constructed in step (2). So, protocol B solves **SPT**. What is the cost ? Since *Shout* uses exactly $2m$ messages, we have

$$M[B] = M[A] + 2m$$

In other words, with at most $O(m)$ additional messages, any election protocol can be made to construct a spanning tree; since $\Omega(m)$ messages are needed anyway (Equation 44), this means that

$$\mathcal{M}(\mathbf{SPT/IR}) \leq \mathcal{M}(\mathbf{Elect/IR}) \tag{45}$$

Focus now on a spanning-tree construction algorithm C. Using C as the first step, it is easy to construct an election protocol D where (Exercise 9.87):

$$M[D] = M[C] + O(n)$$

In other words, the message complexity of **Elect** is no more than that of **Elect** plus at most another $O(n)$ messages; since election requires more than $O(n)$ messages anyway (Property 8.17), this means that

$$\mathcal{M}(\mathbf{Elect/IR}) \leq \mathcal{M}(\mathbf{SPT/IR}) \tag{46}$$

Combining Equations 45 and 46, we have not only that the problems are *computationally equivalent*

$$\mathbf{Elect}(IR) \equiv \mathbf{SPT}(IR) \tag{47}$$

but also that they have the same complexity:

$$\mathcal{M}(\textbf{Elect/IR}) = \mathcal{M}(\textbf{SPT/IR}) \tag{48}$$

Using similar arguments, it is possible to establish the computational and complexity equivalence of election with several other problems (e.g., see Exercise 9.88).

9 Exercises, Problems, and Answers

9.1 Exercises

Exercise 9.1 *Modify protocol* MinF-Tree *(presented in Section* **??***) so to implement strategy* Elect Minimum Initiator *in a tree. Prove its correctness and analyze its costs. Show that, in the worst case, it uses* $3n + k_\star - 4 \leq 4n - 4$ *messages.*

Exercise 9.2 *Design an efficient single-initiator protocol to find the minimum value in a ring. Prove its correctness and analyze its costs.*

Exercise 9.3 *Show that the time costs of protocol* All the Way *will be at most* $2n - 1$*. Determine also the minimum cost and the condition which will cause it.*

Exercise 9.4 *Modify protocol* All the Way *so to use strategy* Elect Minimum Initiator*.*

Exercise 9.5 *Modify protocol* AsFar *so to use strategy* Elect Minimum Initiator*. Determine the average number of messages assuming that any subset of* k_* *entities is equally likely to be the initiators.*

Exercise 9.6 *Expand the rules of protocol* Stages *described in Section 3.4, so to enforce message ordering.*

Exercise 9.7 *Show that, in protocol* Stages*, there will be at most one enqueued message per closed port.*

Exercise 9.8 *Prove that, in protocol* Stages with Feedback*, the minimum distance between two candidates in stage* i *is* $d(i) \geq 2^{i-1}$*.*

Exercise 9.9 *Show an initial configuration for* $n = 8$ *in which protocol* Stages *will require the most messages. Describe how to construct the "worst configuration" for any* n*.*

Exercise 9.10 *Determine the ideal time complexity of protocol* Stages*.*

Exercise 9.11 *Modify protocol* Stages *using the* min-max *approach discussed in Section 3.7.3. Prove its correctness. Show that its message costs are unchanged.*

Exercise 9.12 *Write the rules of protocol* Stages* *described in Section 3.4.*

Exercise 9.13 *Assume that in* Stages* candidate x *in stage* i *receives a message* $M*$ *with stage* $j > i$. *Prove that, if* x *survives, then* $id(x)$ *is smaller not only of* $id*$ *but also of the ids in the messages "jumped over" by* $M*$.

Exercise 9.14 *Show that protocol* Stages* *correctly terminates.*

Exercise 9.15 *Prove that the message and time costs of* Stages* *are no worse that those of* Stages. *Produce an example in which the costs of* Stages* *are actually* smaller.

Exercise 9.16 *Write the rules of protocol* Stages with Feedback *assuming* message ordering.

Exercise 9.17 *Derive the ideal time complexity of protocol* Stages with Feedback.

Exercise 9.18 *Write the rules of protocol* Stages with Feedback *enforcing* message ordering.

Exercise 9.19 *Prove that, in protocol* Stages with Feedback, *the number of ring segments where no feedback will be transmitted in stage* i *is* n_{i+1}.

Exercise 9.20 *Prove that, in protocol* Stages with Feedback, *the minimum distance between two candidates in stage* i *is* $d(i) \geq 3^{i-1}$.

Exercise 9.21 *Give a more accurate estimate of the message costs of protocol* Stages with Feedback.

Exercise 9.22 *Show an initial configuration for* $n = 9$ *in which protocol* Stages with Feedback *will require the most stages. Describe how to construct the "worst configuration" for any* n.

Exercise 9.23 *Modify protocol* Stages with Feedback *using the* min-max *approach discussed in Section 3.7.3. Prove its correctness. Show that its message costs are unchanged.*

Exercise 9.24 *Implement the* alternating step *strategy under the same restrictions and with the same cost of protocol* Alternate *but without closing any port.*

Exercise 9.25 *Determine initial configurations which will force protocol* Alternate *to use* k *steps when* $n = F_k$.

Exercise 9.26 *Show that the worst case number of steps of protocol* Alternate *is* achievable *for* every $n > 4$.

Exercise 9.27 *Determine the ideal time complexity of protocol* Alternate.

Exercise 9.28 *Modify protocol* Alternate *using the* min-max *approach discussed in Section 3.7.3. Prove its correctness. Show that its message costs are unchanged.*

Exercise 9.29 *Show the step-by-step execution of* Stages *and of* UniStages *in the ring of Figure 3. Indicate, for each step, the values know at the candidates.*

Exercise 9.30 *Determine the ideal time complexity of protocol* UniStages.

Exercise 9.31 *Modify protocol* UniStages *using the* min-max *approach discussed in Section 3.7.3. Prove its correctness. Show that its message costs are unchanged.*

Exercise 9.32 *Design an exact simulation of* Stages with Feedback *for unidirectional rings. Analyze its costs.*

Exercise 9.33 *Show the step-by-step execution of* Alternate *and of* UniAlternate *in the ring of Figure 3. Indicate, for each step, the values know at the candidates.*

Exercise 9.34 *Without changing its message cost, modify protocol* UniAlternate *so it does not require Message Ordering.*

Exercise 9.35 *Prove that the ideal time complexity of protocol* UniAlternate *is* $O(n)$.

Exercise 9.36 *Modify protocol* UniAlternate *using the* min-max *approach discussed in Section 3.7.3. Prove its correctness. Show that its message costs are unchanged.*

Exercise 9.37 *Prove that, in protocol* MinMax, *if a* candidate x *survives an even stage* i, *its predecessor* $l(i, x)$ *becomes* defeated.

Exercise 9.38 *Show that the worst case number of steps of protocol* MinMax *is* achievable.

Exercise 9.39 *Modify protocol* MinMax *so it does not require Message Ordering. Implement your modification, and throughly test your implementation.*

Exercise 9.40 *For protocol* MinMax, *consider the configuration depicted in Figure 32. Prove that, once envelope* $(11, 3)$ *reaches the* defeated *node* z, z *can determine that* 11 *will survive this stage.*

Exercise 9.41 *Write the rules of Protocol* MinMax+ *assuming message ordering.*

Exercise 9.42 *Write the rules of Protocol* MinMax+ *without assuming message ordering.*

Exercise 9.43 *Prove Property 3.1.*

Exercise 9.44 *Prove that, in protocol* MinMax+, *if an envelope with value* v *reaches an even stage* $i + 1$, *it saves at least* F_i *messages in stage* i *with respect to* MinMax. (Hint: use Property 3.1.)

Exercise 9.45 *Prove that, even if the entities know* n, $ave_A(I|n \text{ known}) \geq (\frac{1}{4} - \epsilon)\ n \log n$ *for any election protocol* A *for unidirectional rings.*

Exercise 9.46 *Prove that,in bidirectional rings,* $ave_A(I) \geq \frac{1}{2}\, nH_n$ *for any election protocol* A.

Exercise 9.47 *Prove that, even if the entities know* n, $ave_A(I|n \text{ known}) \geq \frac{1}{2} n \log n$ *for any election protocol* A *for unidirectional rings.*

Exercise 9.48 *Determine the exact complexity of Wake-Up in a mesh of dimensions* $a \times b$.

Exercise 9.49 *Show how to broadcast from a corner of a mesh dimensions* $a \times b$ *with less than* $2n$ *messages.*

Exercise 9.50 *In Protocol* ElectMesh*, in the first stage of the election process, if an interior node receives an election message, it will reply to the sender "I am in the interior", so that no subsequent election messages are sent to it. Explain why it is possible to achieving the same goal* without *sending those replies.*

Exercise 9.51 *Consider the following simple modification to Protocol* ElectMesh*: when sending a wake-up message, a node includes the information of whether it is an internal, a border or a corner node. Then, during the first stage of the election, a border node uses this information if possible to send the election message only along the outer ring (it might not be possible !). Show that the protocol so modified uses at most* $4(a+b) + 5n + k_\star - 32$ *messages.*

Exercise 9.52 Broadcasting in Oriented Mesh. *Design a protocol that allows to broadcast in an oriented mesh using* $n-1$ *messages regardless of the location of the initiator.*

Exercise 9.53 Traversal in Oriented Mesh. *Design a protocol which allows to traverse an oriented mesh using* $n-1$ *messages regardless of the location of the initiator.*

Exercise 9.54 Wake-Up in Oriented Mesh. *Design a protocol which allows to wake-up all the entities in an oriented mesh using less than* $2n$ *messages regardless of the location and the number of the initiators.*

Exercise 9.55 *Show that the effect of rounding up* α^i *does not affect the order of magnitude of the cost of Protocol* MarkBorder *derived in Section 4.2.1* [Hint : Show that it amounts to at most 8 extra messages per *candidate* per stage with an insignificant change in the bound on the number of *candidates* in each stage.]

Exercise 9.56 *Show that the ideal time of protocol* MarkBorder *can be as bad as* $O(n)$.

Exercise 9.57 $\star\star$ **Improving Time in Tori.** *Modify Protocol* MarkBorder *so that the time complexity is* $O(\sqrt{n})$ *without increasing the message complexity. Ensure that the modified protocol is correct.*

Exercise 9.58 $\star$ **Election in Rectangular Torus.** *Modify Protocol* MarkBorder *so that it elects a leader in a rectangular torus of dimension* $l \times w$ $(l \leq w)$*, using* $\Theta(n + l \log l/w)$ *messages.*

Exercise 9.59 *Determine the cost of electing a leader in an oriented hypercube if in protocol* HyperElect *the propagation of the Match messages is done by broadcasting in the appropriate subcube instead of "compressing the address".*

Exercise 9.60 *Prove that in protocol* HyperElect *the distance* $d(j-1,j)$ *between* $w^{j-1}(z)$ *and* $w^j(z)$*; is at most* j*.*

Exercise 9.61 *Prove Lemma 5.1; that is, that during the execution of protocol* HyperElect*, the only duelists in stage* i *are the entities with the smallest id in one of the hypercubes of dimension* $i-1$ *in* $H_{k:i-1}$*.*

Exercise 9.62 *Show that the time complexity of Protocol* HyperFlood *is* $O(\log^3 N)$*.*

Exercise 9.63 $\star\star$ *Prove that it is possible to elect a leader in a hypercube using* $O(n)$ *messages with* any *sense of direction.* Hint: use long messages.

Exercise 9.64 *Prove that, in the strategy* CompleteElect *outlined in section 6.1, the territory of any two candidates in the same stage have no nodes in common.*

Exercise 9.65 *Prove that the strategy* CompleteElect *outlined in section 6.1 solves the election problem.*

Exercise 9.66 *Determine the cost of the strategy* CompleteElect *described in section 6.1 in the worst case. (Hint: consider how many candidates there can be at level* i*.)*

Exercise 9.67 *Analyze the ideal time cost of protocol* CompleteElect *described in section 6.1.*

Exercise 9.68 *Design an election protocol for complete graphs which, like* CompleteElect*, uses* $O(n \log n)$ *messages but uses only* $O(n/\log n)$ *time in the worst case.*

Exercise 9.69 *Generalize the answer to Exercise 9.68. Design an election protocol for complete graphs which, for any* $\log n \leq k \leq n$*, uses* $O(nk)$ *messages and* $O(n/k)$ *time in the worst case.*

Exercise 9.70 *Prove that all the rings* $R(2), \ldots, R(k)$ *where messages are sent by protocol* Kelect *do not have links in common.*

Exercise 9.71 *Write the code for, implement, and test protocol* Kelect*.*

Exercise 9.72 $\star$ *Consider using the ring protocol* Alternate *instead of* Stages *in* Kelect*. Determine what will be the cost in this case.*

Exercise 9.73 $\star\star$ *Determine the average message costs of protocol* Kelect*.*

Exercise 9.74 $\star$ *Show how to elect a leader in a complete network with $O(n \log n)$ messages in the worst case but only $O(n)$ on the average.*

Exercise 9.75 $\star\star$ *Prove that it is possible to elect a leader in a complete graph using $O(n)$ messages with any sense of direction.*

Exercise 9.76 *Show how to elect a leader in the chordal ring $C_n\langle 1, 2, 3, 4..., t\rangle$ with $O(n + \frac{n}{t} \log \frac{n}{t}$ messages.*

Exercise 9.77 *Prove that, in chordal ring C_n^t electing a leader requires at lest $\Omega(n + \frac{n}{t} \log \frac{n}{t}$ messages in the worst case.* Hint: Reduce the problem to that of electing a leader on a ring of size n/t.

Exercise 9.78 *Show how to elect a leader in the* double cube *$C_n\langle 1, 2, 4, 8..., 2^{\lceil \log n \rceil}\rangle$ with $O(n)$ messages.*

Exercise 9.79 *Consider a merger message from city A arriving at neighbouring city B along merge link (a, b) in protocol* Mega-Merger. *Prove that, if we reverse the logical direction of the links on the path from $D(A)$ to the exit point a, and direct towards B the merge link, the union of A and B will be rooted in the downtown of A.*

Exercise 9.80 *District x is involved in the computing of the merge link of its city X. In the meanwhile, it receives a* Let-us-Merge *message from the unused link (x, y). From the message, x finds out that $level(Y) < level(X)$, and thus Y must be absorbed in X. Should Y (now becoming part of X) be included in the computation of the merge link of X ? Why ?*

Exercise 9.81 *District b of B has just received a* Let-us-Merge *message from a along merge link (a, b). From the message, b finds out that $level(A) > level(B)$; thus, it postpones the request. In the meanwhile, the downtown $D(B)$ chooses (a, b) as its merge link. Can this situation occur ? If so, what will happen ? If not, why ?*

Exercise 9.82 *Find a way to avoid notification of termination by the downtown of the megacity in protocol* Mega-Merger. *(Hint. Show that, by the time the downtown understands that the mega-merger is completed, all other districts already know that their execution of the protocol is terminated)*

Exercise 9.83 Time Costs. *Show that protocol* Mega-Merger *uses at most $O(n \log n)$ ideal time units.*

Exercise 9.84 *Prove that, in the* Yo-Yo *protocol, during an iteration, no sink or internal node will become a source.*

Exercise 9.85 *Modify the* Yo-Yo *protocol so that, upon termination, a spanning-tree rooted in the leader has been constructed. Achieve this goal* without *any additional messages.*

Exercise 9.86 *Prove that to solve* **SPT** *under* **IR**, *a message must be sent on every link.*

Exercise 9.87 *Show how to transform a spanning-tree construction algorithm C so to elect a leader with at most $O(n)$ additional messages.*

Exercise 9.88 *Prove that, under* **IR**, *the problem of finding the smallest of the entities' values is computationally equivalent to electing a leader, and has the same message complexity.*

9.2 Problems

Problem 9.1 Josephus Problem. *Consider the following set of electoral rules. In stage i, a candidate x sends its id and receives the id from its two neighbouring candidates, $r(i,x)$ and $l(i,x)$: x does not survive this stage if and only if its id is larger than both received ids. Analyze the corresponding protocol* Josephus*, determining in particular the number of stages and the total number of messages both in the worst and in the average case. Analyze and discuss its time complexity.*

Problem 9.2 $\star$ Alternating Steps. *Design a* conflict resolution *mechanism for the* alternating steps *strategy to cope lack of orientation in the ring. Analyze the complexity of the resulting protocol*

Problem 9.3 $\star\star$ Better Stages. *Construct a protocol based on electoral stages which guarantees $n_i \leq \frac{n_{i-1}}{b}$ with cn messages transmitted in each stage, where $\frac{c}{\log b} < 1.89$.*

Problem 9.4 $\star$ Bidirectional *MinMax*. *Design a bidirectional version of* MinMax *with the same costs.*

Problem 9.5 $\star\star$ Distances in *MinMax+*. *In computing the cost of protocol* MinMax+ *we have used $dis(i) = F_{i+2}$. Determine what will be the cost if we use $dis(i) = 2^i$ instead.*

Problem 9.6 $\star\star$ *MinMax+* Variations. *In protocol* MinMax+ *we use "promotion by distance" only in the even stages, and "promotion by witness" only in the odd stages. Determine what would happen if we use*
(1) only "promotion by distance" but in every stage;
(2) only "promotion by witness" but in every stage;
(3) "promotion by distance" in every stage, and "promotion by witness" only in odd stages;
(4) "promotion by witness" in every stage, and "promotion by distance" only in even stages;
(5) both "promotion by distance" and "promotion by witness" in every *stage.*

Problem 9.7 $\star\star$ Bidirectional Oriented Rings. *Prove or disprove that there is an efficient protocol for bidirectional oriented rings that cannot be used nor simulated in unidirectional rings, nor in general bidirectional ones with the same or better costs.*

Problem 9.8 $\star$ Unoriented Hypercubes. *Design a protocol that can elect a leader in a hypercube with arbitrary labelling using $O(n \log\log n)$ messages. Implement and test your protocol.*

Problem 9.9 $\star\star\star$ Linear Election in Hypercubes. *Prove or disprove that it is possible to elect a leader in an hypercube in $O(n)$ messages even when it is not oriented.*

Problem 9.10 $\star$ Oriented Cube-Connected-Cycles. *Design an election protocol for an oriented* CCC *using $O(n)$ messages. Implement and test your protocol.*

Problem 9.11 Oriented Butterfly. *Design an election protocol for an oriented* butterfly*. Determine its complexity. Implement and test your protocol.*

Problem 9.12 ⋆⋆ **Minimal Chordal Ring.** *Find a chordal ring with $k = 2$ where it is possible to elect a leader with $O(n)$ messages.*

Problem 9.13 ⋆⋆ **Unlabelled Chordal Rings.** *Show how to elect a leader in the chordal ring of Problem 9.12 with $O(n)$ messages even if the edges are arbitrarily labelled.*

Problem 9.14 ⋆ **Improved Time** *Show how to elect a leader using $O(m+n \log n)$ messages but only $O(n)$ ideal time units.*

Problem 9.15 ⋆ **Optimal Time.** *Show how to elect a leader in $O(d)$ time using at most $O(m \log d)$ messages.*

9.3 Answers to Exercises

Answer to Exercise 9.21
The size of the areas where no feedback is sent in stage i can vary from one another, from stage to stage, from execution to execution. We can still have an estimate of their size. In fact, the distance d_i between two candidates in stage i is $d(i) \geq 3^{i-1}$ (Exercise 9.20. Thus, the total number of message transmissions caused in stage i by the feedback will be at most $n - n_{i+1}3^{i-1}$, yielding a total of at most $3n - \sum_{i=1}^{\lceil log_3 n \rceil} n_{i+1}3^{i-1}$ messages.

Answer to Exercise 9.44
Let $h_j(a)$ denote the *candidate* that originated message (a, j). Consider a message $(v, i+1)$ and its originator z= $h_{i+1}(v)$; this message was sent after receiving (v, i) originated by $x = h_i(v)$.
Let $y = h_i(u)$ be the first *candidate* after x in the ring in stage i, and (u, i) the message it originated. Since v survives this stage, which is odd (i.e., *min*), then it must be that $v < u$. Message (v, i) travels from x towards y; upon receiving (v, i), node z in this interval will generate $(v, i+1)$. Now z cannot be after node $h_{i-1}(u)$ in the ring, because by rule (IV) $w = h_{i-1}(u)$ would immediately generate $(v, i+1)$ after receiving (v, i). In other words, either $z = w$ or z is before w. Thus we save at least $d(z, y) \geq d(w, y) = d(h_{i-1}(u), h_i(u)) \geq F_i$, where the last inequality is by Property 3.1.

Partial Answer to Exercise 9.66 Consider a *captured* node y which receives an attack after the other, say from a candidates x_1 in level i. According to the strategy, y will send a Warning to its owner z to inform it of this attack, and wait for a reply; depending on the reply, it will notify x_1 of whether the attack was successful (in which case y will be captured by x_1) or not. Assume now that, while waiting, y receives an attack after the other, say from candidates $x_2, \ldots, x_k$ in that order, all in the same level i. According to the strategy, y will issue a Warning to its owner z for each of them. Observe now that if $id(z) > id(x_1) > \ldots > id(x_k)$, each of these attacks will be successful ! and y will in turn be captured by all those candidates.

10 Bibliographical Notes

Election in a ring network is one of the first problems studied in distributed computing from an algorithmic point of view. The first solution protocol, *All the Way*, is due to Gerard LeLann [**?**] proposed for unidirectional rings. Also for unidirectional rings, protocol *AsFar* was developed by Ernie Chang and Rosemary Roberts [**?**]. The probabilistic bidirectional version *ProbAsFar* was proposed and analyzed by Ephraim Korach, Doron Rotem and Nicola Santoro [**?**]. Hans Boadlander and Jan van Leeuwen later showed how to make it deterministic and provided further analysis [**?**]; the exact asymptotic average value has been derived by Chistian Lavault [**?**].

The idea beyond the first $\Theta(n \log n)$ worst case protocol, *Control*, is due to Dan Hirschberg and Sinclair [**?**]. Protocol *Stages* was designed by Franklin [**?**]; the more efficient *Stages with Feedback* was developed by Ephraim Korach, Doron Rotem and Nicola Santoro [**?**].

The first $\Theta(n \log n)$ worst case protocol for unidirectional rings, *UniStages*, was designed by Danny Dolev, Maria Klawe, and Mikey Rodeh [**?**]. The more efficient *MinMax* is due to Gary Peterson [**?**]. The even more efficient protocol *MinMax+* has been designed by Lisa Higham and Theresa Przytycka [**?**]. Bidirectional versions of *MinMax* with the same complexity as the original (Problem 9.4) have been independently designed by Shlomo Moran, M. Shalom, and Shmuel Zaks [**?**], and by Jan van Leeuwen and Richard Tan [**?**].

The lower bound for unidirectional rings is due to Jan Pachl, Doron Rotem, and Ephraim Korach [**?**]. James Burns developed the first lower bound for bidirectional rings [**?**]. The lower bounds when n is known known (Exercises 9.45 and 9.47), as well as others, are due to Hans Boadlander [**?**, **?**, **?**].

The $O(n)$ election protocol for *tori* was designed by Gary Peterson [**?**], and later refined for unoriented tori by Bernard Mans [**?**].

The quest for a $O(n)$ election protocol for *hypercubes* with dimensional labellings was solved independently by S. Robbins and K.A. Robbins [**?**], Paola Flocchini and Bernard Mans [**?**], and Gerard Tel [**?**] (who, in his book, credits only himself). Stefan Dobrev [**?**] has designed a protocol that allows $O(n)$ election in hypercubes with *any* sense of direction, not just the dimensional labelling (Exercise 9.63). The protocol for *unoriented hypercubes* has been designed by Stefan Dobrev and Peter Ruzicka [**?**].

The first optimal $\Theta(n \log n)$ protocol for *complete networks* was developed by Pierre Humblet [**?**]; an optimal protocol that requires $O(n)$ messages on the average (Exercise 9.74) was developed by M. Chan and Francis Chin [**?**]. The lower-bound is due to Ephraim Korach, Shlomo Moran, and Shmuel Zaks [**?**] who also designed another optimal protocol. The optimal protocol *CompleteElect*, reducing the $O(nlogn)$ time complexity to $O(n)$, was designed by Yeuda Afek and Eli Gafni [**?**]; the same bounds were independently achieved by Gary Peterson [**?**]. The time complexity has been later reduced to $O(\frac{n}{\log n})$ without increasing the message costs (Exercise 9.68) by Gurdip Singh [**?**].

The fact that a chordal labelling allows to fully exploit the communication power of the complete graph was observed by Mike Loui, T.A. Matsushita, and Doug West who developed

the first $O(n)$ protocol for such a case [**?**]; G.H. Masapati and Hasan Ural [**?**] later showed how the use of few preprocessing steps can reduce the multiplicative constant. Stefan Dobrev [**?**] has designed a protocol that allows $O(n)$ election in complete networks with *any* sense of direction, not just the chordal labelling (Exercise 9.75).

Election protocols for *chordal rings*, including the doublecube, were designed and analyzed by Hagit Attiya, Jan van Leeuwen, Nicola Santoro, and Shmuel Zaks [**?**]. The quest for the smallest cord structure has seen k being reduced from $O(\log n)$ first to $O(\log \log n)$ by T.Z. Kalamboukis and S.L. Mantzaris [**?**], then to $O(\log \log \log n)$ by Yi Pan [**?**], and finally to $O(1)$ (Problem 9.12) by Andreas Fabri and Gerard Tel [unpublished]. The observation that, in such a chordal ring, election can be done in $O(n)$ messages even if the links are arbitrarily labelled (Problem 9.13) is due to Bernard Mans [**?**].

The first $O(m + n \log n)$ election protocol was designed by Robert Gallager [**?**]. Some of the ideas developed there were later used in *MegaMerger*, developed by Robert Gallager, Pierre Humblet, and P. Spira, that actually constructs a min-cost spanning tree [**?**]. Protocol *YO-YO* was designed by Nicola Santoro; the proof that it requires at most $O(logn)$ stages is due to Gerard Tel. The $\Omega(m + n \log n)$ lowerbound and the message complexity equivalence results were first observed by Nicola Santoro [**?**].

The $O(n \log n)$ time complexity of *MegaMerger* has been reduced first to $O(n \log^* n)$ by M. Chan and Francis Chin [**?**], and then to $O(n)$ (Problem 9.14) by Baruch Awerbuch [] without increasing the message complexity. It has been further reduced to $\Theta(d)$ (Problem 9.15) by Hasame Abu-Amara and A. Kanevsky but at the expenses of a $O(m \log d)$ message cost [**?**].