Chapter 3

Leader Election in a Synchronous Ring

In this chapter, we present the first problem to be solved using the synchronous model of Chapter 2: the problem of *electing a unique leader* process from among the processes in a network. For starters, we consider the simple case where the network digraph is a ring.

This problem originally arose in the study of local area *token ring* networks. In such a network, a single "token" circulates around the network, giving its current owner the sole right to initiate communication. (If two nodes in the network were to attempt simultaneously to communicate, the communications could interfere with one another.) Sometimes, however, the token may be lost, and it becomes necessary for the processes to execute an algorithm to regenerate the lost token. This regeneration procedure amounts to electing a leader.

3.1 The Problem

We assume that the network digraph G is a ring consisting of n nodes, numbered 1 to n in the clockwise direction (see Figure 3.1). We often count mod n, allowing 0 to be another name for process n, $n + 1$ another name for process 1, and so on. The processes associated with the nodes of G do not know their indices, nor those of their neighbors; we assume that the message-generation and transition functions are defined in terms of local, relative names for the neighbors. However, we do assume that each process is able to distinguish its clockwise neighbor from its counterclockwise neighbor. The requirement is that, eventually, exactly one process should output the decision that it is the leader, say by changing a special *status* component of its state to the value *leader*. There are several versions of

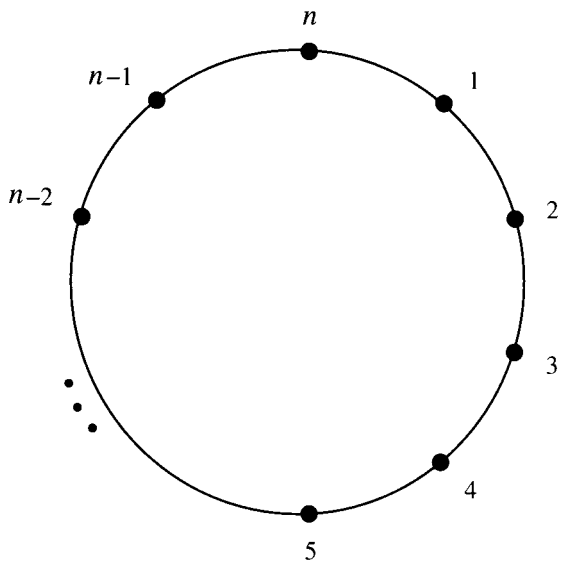

Figure 3.1: A ring of processes.

the problem:

1. It might also be required that all non-leader processes eventually output the fact that they are not the leader, say by changing their *status* components to the value *non-leader*.

2. The ring can be either *unidirectional* or *bidirectional*. If it is unidirectional, then each edge is directed from a process to its clockwise neighbor, that is, messages can only be sent in a clockwise direction.

3. The number n of nodes in the ring can be either known or unknown to the processes. If it is known, it means that the processes only need to work correctly in rings of size n, and thus they can use the value n in their programs. If it is unknown, it means that the processes are supposed to work in rings of different sizes. Therefore, they cannot use information about the ring size.

4. Processes can be identical or can be distinguished by each starting with a *unique identifier* (*UID*) chosen from some large totally ordered space of identifiers such as the positive integers, $\mathbb{N}^+$. We assume that each process's UID is different from each other's in the ring, but that there is no constraint on which UIDs actually appear in the ring. (For instance, they do not have to be consecutive integers.) These identifiers can be restricted to be

manipulated only by certain operations, such as comparisons, or they can admit unrestricted operations.

3.2 Impossibility Result for Identical Processes

A first easy observation is that if all the processes are identical, then this problem cannot be solved at all in the given model. This is so even if the ring is bidirectional and the ring size is known to the processes.

Theorem 3.1 *Let A be a system of n processes, $n > 1$, arranged in a bidirectional ring. If all the processes in A are identical, then A does not solve the leader-election problem.*

Proof. Suppose that there is such a system A that solves the leader-election problem. We obtain a contradiction. We can assume without any loss of generality that each process of A has exactly one start state. This is so because if each process has more than one start state, we could simply choose any one of the start states and obtain a new solution in which each process has only one start state. With this assumption, A has exactly one execution.

So consider the (unique) execution of A. It is straightforward to verify, by induction on the number r of rounds that have been executed, that all the processes are in identical states immediately after r rounds. Therefore, if any process ever reaches a state where its *status* is *leader*, then all the processes in A also reach such a state at the same time. But this violates the uniqueness requirement. □

Theorem 3.1 implies that the only way to solve the leader-election problem is to break the symmetry somehow. A reasonable assumption derived from what is usually done in practice is that the processes are identical except for a UID. This is the assumption we make in the rest of this chapter.

3.3 A Basic Algorithm

The first solution we present is a fairly obvious one, which we call the *LCR algorithm* in honor of Le Lann, Chang, and Roberts, from whose papers this algorithm is extracted. The algorithm uses only unidirectional communication and does not rely on knowledge of the size of the ring. Only the leader performs an output. The algorithm uses only comparison operations on the UIDs. Below is an informal description of the *LCR* algorithm.

LCR algorithm (informal):

Each process sends its identifier around the ring. When a process receives an incoming identifier, it compares that identifier to its own. If the incoming identifier is greater than its own, it keeps passing the identifier; if it is less than its own, it discards the incoming identifier; if it is equal to its own, the process declares itself the leader.

In this algorithm, the process with the largest UID is the only one that outputs *leader*. In order to make this intuition precise, we give a more careful description of the algorithm in terms of the model of Chapter 2.

LCR algorithm (formal):

The message alphabet M is exactly the set of UIDs.

For each i, the states in $states_i$ consist of the following components:

u, a UID, initially i's UID
send, a UID or *null*, initially i's UID
status, with values in $\{unknown, leader\}$, initially *unknown*

The set of start states $start_i$ consists of the single state defined by the given initializations.

For each i, the message-generation function $msgs_i$ is defined as follows:

send the current value of *send* to process $i + 1$

Actually, process i would use a relative name for process $i+1$, for example, "clockwise neighbor"; we write $i + 1$ because it is simpler. Recall from Chapter 2 that we use the *null* value as a placeholder indicating the absence of a message. So if the value of the *send* component is *null*, this msg_i function does not actually send any message.

For each i, the transition function $trans_i$ is defined by the following pseudocode:

```
send := null
if the incoming message is v, a UID, then
    case
        v > u: send := v
        v = u: status := leader
        v < u: do nothing
    endcase
```

The first line of the transition function definition just cleans up the state from the effects of the preceding message delivery (if any). The rest of the code contains the interesting work—the decision about whether to pass on or discard the incoming UID, or to accept it as permission to become the leader.

This description is written in what should be a reasonably readable programming language, but note that it has a direct translation into a process state machine in the model in Chapter 2. In this translation, each process state consists of a value for each of the variables, and the transitions are describable in terms of changes to the variables. Note that the entire block of code written for the $trans_i$ function is supposed to be executed indivisibly, as part of the processing for a single round.

How do we go about proving formally that the algorithm is correct? Correctness means that exactly one process eventually performs a *leader* output. Let $i_{\max}$ denote the index of the process with the maximum UID, and let $u_{\max}$ denote its UID. It is enough to show that (1) process $i_{\max}$ outputs *leader* by the end of round n, and (2) no other process ever performs such an output. We prove these two properties, respectively, in Lemmas 3.2 and 3.3.

Here and in many other places in the book, we attach the subscript i to a state component name to indicate the instance of that state component belonging to process i. For example, we use the notation u_i to denote the value of state component u of process i. We generally omit the subscripts when writing the process code, however.

Lemma 3.2 *Process i_{max} outputs leader by the end of round n.*

Proof. Note that $u_{\max}$ is the initial value of variable $u_{i_{\max}}$, the variable u at process $i_{\max}$, by the initialization. Also note that the values of the u variables never change (by the code), that they are all distinct (by assumption), and that $i_{\max}$ has the largest u value (by definition of $i_{\max}$). By the code, it suffices to show the following invariant assertion:

Assertion 3.3.1 *After n rounds, $status_{i_{max}} = leader$.*

The normal way to try to prove an invariant such as this one is by induction on the number of rounds. But in order to do this, we need a preliminary invariant that says something about the situation after smaller numbers of rounds. We add the following assertion:

Assertion 3.3.2 *For $0 \leq r \leq n-1$, after r rounds, $send_{i_{max}+r} = u_{max}$.*

(Recall that addition is modulo n.) This assertion says that the maximum value appears in the *send* component at the position in the ring at distance r from $i_{\max}$.

It is straightforward to prove Assertion 3.3.2 by induction on r. For $r = 0$, the initialization says that $send_{i_{\max}} = u_{\max}$ after 0 rounds, which is just what is needed. The inductive step is based on the fact that every node other than $i_{\max}$ accepts the maximum value and places it into its *send* component, since $u_{\max}$ is greater than all the other values.

Having proved Assertion 3.3.2, we use its special case for $r = n - 1$ and one more argument about what happens in a single round to show Assertion 3.3.1. The key fact here is that process $i_{\max}$ accepts $u_{\max}$ as a signal to set its *status* to *leader*. □

Lemma 3.3 *No process other than i_{max} ever outputs the value leader.*

Proof. It is enough to show that all other processes always have *status* = *unknown*. Again, it helps to state a stronger invariant. If i and j are any two processes in the ring, $i \neq j$, define $[i, j)$ to be the set of indices $\{i, i+1, \ldots, j-1\}$, where addition is modulo n. That is, $[i, j)$ is the set of processes starting with i and moving clockwise around the ring up to and including j's counterclockwise neighbor. The following invariant asserts that no UID v can reach any *send* variable in any position between $i_{\max}$ and v's original home i:

> **Assertion 3.3.3** *For any r and any i, j, the following holds. After r rounds, if $i \neq i_{max}$ and $j \in [i_{max}, i)$ then $send_j \neq u_i$.*

Again, it is straightforward to prove the assertion by induction; now the key fact used in the proof is that a non-maximum value does not get past $i_{\max}$. This is because $i_{\max}$ compares the incoming value with $u_{\max}$, and $u_{\max}$ is greater than all the other UIDs.

Finally, Assertion 3.3.3 can be used to show that only process $i_{\max}$ can receive its own UID in a message, and hence only process $i_{\max}$ can output *leader*. □

Lemmas 3.2 and 3.3 together imply the following:

Theorem 3.4 *LCR solves the leader-election problem.*

Halting and *non-leader* outputs. As written, the *LCR* algorithm never finishes its work, in the sense of all the processes reaching a halting state. We can augment each process to include halting states, as described in Section 2.1. Then we can modify the algorithm by allowing the elected leader to initiate a special *report* message to be sent around the ring. Any process that receives the *report* message can halt, after passing it on. This strategy not only allows processes to halt, but could also be used to allow the non-leader processes to output *non-leader*. Furthermore, by attaching the leader's index to the *report* message, this

strategy could also allow all the participating processes to output the identity of the leader. Note that it is also possible for each non-leader node to output *non-leader* immediately after it sees a UID greater than its own; however, this does not tell the non-leader nodes when to halt.

In general, halting is an important property for a distributed algorithm to satisfy; however, it cannot always be achieved as easily as in this case.

Complexity analysis. The time complexity of the basic *LCR* algorithm is n rounds until a leader is announced, and the communication complexity is $O(n^2)$ messages in the worst case. In the halting version of the algorithm, the time complexity is $2n$ and the communication complexity is still $O(n^2)$. The extra time needed for halting and for the *non-leader* announcements is only n rounds, and the extra communication is only n messages.

Transformation. The preceding two remarks describe and analyze a general transformation, from any leader-election algorithm in which only the leader provides output and no process ever halts, to one in which the leader and the non-leaders all provide output and all processes halt. The extra cost of obtaining the extra outputs and the halting is only n rounds and n messages. This transformation works for any combination of our other assumptions.

Variable start times. Note that the *LCR* algorithm works without modification in the version of the synchronous model with variable start times. See Section 2.1 for a description of this version of the model.

Breaking symmetry. In the problem of electing a leader in a ring, the key difficulty is breaking symmetry. Symmetry-breaking is also an important part of many other problems that need to be solved in distributed systems, including *resource-allocation* problems (see Chapters 10–11 and 20) and *consensus* problems (see Chapters 5–7, 12, 21, and 25).

3.4 An Algorithm with $O(n \log n)$ Communication Complexity

Although the time complexity of the *LCR* algorithm is low, the number of messages used by the algorithm seems somewhat high, a total of $O(n^2)$. This might not seem significant, because there is never more than one message on any link at any time. However, in Chapter 2, we discussed why the number of messages is an interesting measure to try to minimize; this is because of the possible network

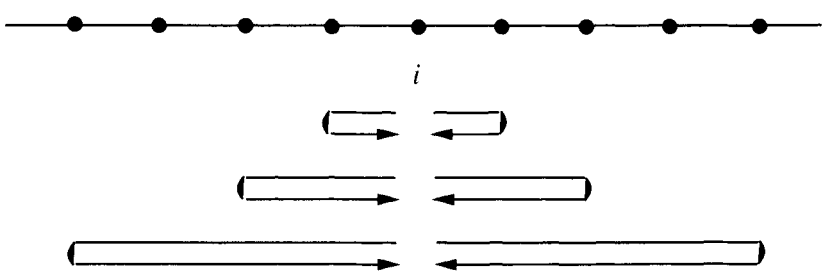

Figure 3.2: Trajectories of successive tokens originating at process i in the *HS* algorithm.

congestion that can result from the total communication load of many concurrently running distributed algorithms. In this section, we present an algorithm that lowers the communication complexity to $O(n \log n)$.

The first published algorithm to reduce the worst-case complexity to $O(n \log n)$ was that of Hirschberg and Sinclair, so we call this algorithm the *HS algorithm.* Again, we assume that only the leader needs to perform an output, though the transformation at the end of Section 3.3 implies that this restriction is not important. Again, we assume that the ring size is unknown, but now we allow bidirectional communication.

As does the *LCR* algorithm, the *HS* algorithm elects the process with the maximum UID. Now every process, instead of sending its UID all the way around the ring as in the *LCR* algorithm, sends it so that it travels some distance away, then turns around and comes back to the originating process. It does this repeatedly, to successively greater distances. The *HS* algorithm proceeds as follows.

***HS* algorithm (informal):**

Each process i operates in phases $0, 1, 2, \ldots$. In each phase l, process i sends out "tokens" containing its UID u_i in both directions. These are intended to travel distance 2^l, then return to their origin i (see Figure 3.2). If both tokens make it back safely, process i continues with the following phase. However, the tokens might not make it back safely. While a u_i token is proceeding in the outbound direction, each other process j on u_i's path compares u_i with its own UID u_j. If $u_i < u_j$, then j simply discards the token, whereas if $u_i > u_j$, then j relays u_i. If $u_i = u_j$, then it means that process j has received its own UID before the token has turned around, so process j elects itself as the leader.

All processes always relay all tokens in the inbound direction.

Now we describe the algorithm more formally. This time, the formalization requires some bookkeeping to ensure that tokens follow the proper trajectories.

For instance, flags are carried by the tokens indicating whether they are travelling outbound or inbound. Also, hop counts are carried with the tokens to keep track of the distances they must travel in the outbound direction; this allows the processes to figure out when the directions of the tokens should be reversed. Once the algorithm is formalized in this way, a correctness argument of the sort given for *LCR* can be provided.

_HS_ algorithm (formal):

The message alphabet M is the set of triples consisting of a UID, a *flag* value in $\{out, in\}$, and a positive integer *hop-count*.

For each i, the states in $states_i$ consist of the following components:

u, of type UID, initially i's UID
send+, containing either an element of M or *null*,
    initially the triple consisting of i's UID, *out*, and 1
send−, containing either an element of M or *null*,
    initially the triple consisting of i's UID, *out*, and 1
status, with values in $\{unknown, leader\}$, initially *unknown*
phase, a nonnegative integer, initially 0

The set of start states $start_i$ consists of the single state defined by the given initializations.

For each i, the message-generation function $msgs_i$ is defined as follows:

send the current value of *send+* to process $i + 1$
send the current value of *send−* to process $i - 1$

For each i, the transition function $trans_i$ is defined by the following pseudocode:

send+ := *null*
send− := *null*
if the message from $i - 1$ is (v, out, h) then
    case
        $v > u$ and $h > 1$: *send+* := $(v, out, h - 1)$
        $v > u$ and $h = 1$: *send−* := $(v, in, 1)$
        $v = u$: *status* := *leader*
    endcase
if the message from $i + 1$ is (v, out, h) then
    case
        $v > u$ and $h > 1$: *send−* := $(v, out, h - 1)$
        $v > u$ and $h = 1$: *send+* := $(v, in, 1)$
        $v = u$: *status* := *leader*
    endcase
if the message from $i - 1$ is $(v, in, 1)$ and $v \neq u$ then

```
    send+ := (v, in, 1)
if the message from i + 1 is (v, in, 1) and v ≠ u then
    send− := (v, in, 1)
if the messages from i − 1 and i + 1 are both (u, in, 1) then
    phase := phase + 1
    send+ := (u, out, 2^phase)
    send− := (u, out, 2^phase)
```

As before, the first two lines just clean up the state. The next two pieces of code describe the handling of outbound tokens: tokens with UIDs that are greater than u_i are either relayed or turned around, depending on the *hop-count*, and the receipt of u_i causes i to elect itself the leader. The next two pieces of code describe the handling of inbound tokens: they are simply relayed. (A trivial *hop-count* of 1 is used for inbound tokens.) If process i receives both of its own tokens back, then it goes on to the next phase.

Complexity analysis. We first analyze the communication complexity. Every process sends out a token in phase 0; this is a total of $4n$ messages for the token to go out and return, in both directions. For $l > 0$, a process sends a token in phase l exactly if it receives both its phase $l-1$ tokens back. This is exactly if it has not been "defeated" by another process within distance 2^{l-1} in either direction along the ring. This implies that within any group of $2^{l-1}+1$ consecutive processes, at most one goes on to initiate tokens at phase l. This can be used to show that at most

$$\left\lfloor \frac{n}{2^{l-1}+1} \right\rfloor$$

processes altogether initiate tokens at phase l. Then the total number of messages sent out at phase l is bounded by

$$4\left(2^l \cdot \left\lfloor \frac{n}{2^{l-1}+1} \right\rfloor\right) \leq 8n.$$

This is because phase l tokens travel distance 2^l. Again, the factor of 4 is derived from the fact that the token is sent out in both directions—clockwise and counterclockwise—and that each outbound token must turn around and return.

The total number of phases that are executed before a leader is elected and all communication stops is at most $1 + \lceil \log n \rceil$ (including phase 0), so the total number of messages is at most $8n(1 + \lceil \log n \rceil)$, which is $O(n \log n)$, with a constant factor of approximately 8.

The time complexity for this algorithm is just $O(n)$. This can be seen by noting that the time for each phase l is $2 \cdot 2^l = 2^{l+1}$ (for the tokens to go out and

return). The final phase takes time n—it is an incomplete phase, with tokens only travelling outbound. The next-to-last phase is phase $l = \lceil \log n \rceil - 1$, and its time complexity is at least as great as the total time complexity of all the preceding phases. Thus, the total time complexity of all but the final phase is at most

$$2 \cdot 2^{\lceil \log n \rceil}.$$

It follows that the total time complexity is at most $3n$ if n is a power of 2, and $5n$ otherwise. The rest of the details are left as an exercise.

Variable start times. The *HS* algorithm works without modification in the version of the synchronous model with variable start times.

3.5 Non-Comparison-Based Algorithms

We next consider the question of whether it is possible to elect a leader with fewer than $O(n \log n)$ messages. The answer to this problem, as we shall demonstrate shortly with an impossibility result—a lower bound of $\Omega(n \log n)$—is negative. That result, however, is valid only in the case of algorithms that manipulate the UIDs using comparisons only. (*Comparison-based algorithms* are defined in Section 3.6 below.)

In this section, we allow the UIDs to be positive integers and permit them to be manipulated by general arithmetic operations. For this case, we give two algorithms, the *TimeSlice algorithm* and the *VariableSpeeds algorithm*, each of which uses only $O(n)$ messages. The existence of these algorithms implies that the lower bound of $\Omega(n \log n)$ cannot be proved for the general case.

3.5.1 The *TimeSlice* Algorithm

The first of these algorithms uses the strong assumption that the ring size n is known to all the processes, but only assumes unidirectional communication. In this setting, the following simple algorithm, which we call the *TimeSlice algorithm*, works. It elects the process with the minimum UID.

Note that this algorithm uses synchrony in a deeper way than do the *LCR* and *HS* algorithms. Namely, it uses the non-arrival of messages (i.e., the arrival of *null* messages) at certain rounds to convey information.

***TimeSlice* algorithm:**

Computation proceeds in phases 1, 2, ..., where each phase consists of n consecutive rounds. Each phase is devoted to the possible circulation, all the way around the ring, of a token carrying a particular UID. More

specifically, in phase v, which consists of rounds $(v-1)n+1, \ldots, vn$, only a token carrying UID v is permitted to circulate.

If a process i with UID v exists, and round $(v-1)n+1$ is reached without i having previously received any non-*null* messages, then process i elects itself the leader and sends a token carrying its UID around the ring. As this token travels, all the other processes note that they have received it, which prevents them from electing themselves as leader or initiating the sending of a token at any later phase.

With this algorithm, the minimum UID $u_{\min}$ eventually gets all the way around, which causes its originating process to become elected. No messages are sent before round $(u_{\min}-1)n+1$, and no messages are sent after round $u_{\min} \cdot n$. The total number of messages sent is just n. If we prefer to elect the process with the maximum UID rather than the process with the minimum, we can simply let the minimum send a special message around after it is discovered in order to determine the maximum. The communication complexity is still $O(n)$.

The good property of the *TimeSlice* algorithm is that the total number of messages is n. Unfortunately, the time complexity is about $n \cdot u_{\min}$, which is an unbounded number, even in a fixed-size ring. This time complexity limits the practicality of the algorithm; it is only useful in practice for small ring networks in which UIDs are assigned from among the small positive integers.

3.5.2 The *VariableSpeeds* Algorithm

The *TimeSlice* algorithm shows that $O(n)$ messages are sufficient in the case of rings in which processes know n, the size of the ring. But what if n is unknown? It turns out that in this case, also, there is an $O(n)$ message algorithm, which we call the *VariableSpeeds algorithm* for reasons that will become apparent in a moment. The *VariableSpeeds* algorithm uses only unidirectional communication.

Unfortunately, the time complexity of the *VariableSpeeds* algorithm is even worse than that of the *TimeSlice* algorithm: $O(n \cdot 2^{u_{\min}})$. Clearly, no one would even think of using this algorithm in practice! The *VariableSpeeds* algorithm is what we call a *counterexample algorithm.* A *counterexample algorithm* is one whose main purpose is to show that a conjectured impossibility result is false. Such an algorithm is generally not interesting by itself—it is neither practical nor particularly elegant from a mathematical viewpoint. However, it does serve to show that an impossibility result cannot be proved.

Here is the algorithm.

***VariableSpeeds* algorithm:**

Each process i initiates a token, which travels around the ring, carrying

the UID u_i of the originating process i. Different tokens travel at different rates. In particular, a token carrying UID v travels at the rate of 1 message transmission every 2^v rounds, that is, each process along its path waits 2^v rounds after receiving the token before sending it out.

Meanwhile, each process keeps track of the smallest UID it has seen and simply discards any token carrying an identifier that is larger than this smallest one.

If a token returns to its originator, the originator is elected.

As for the *TimeSlice* algorithm, the *VariableSpeeds* algorithm guarantees that the process with the minimum UID is elected.

Complexity analysis. The *VariableSpeeds* algorithm guarantees that by the time the token carrying the smallest identifier $u_{\min}$ gets all the way around the ring, the second smallest identifier could only get at most halfway around, the third smallest could only get at most a quarter of the way around, and in general, the kth smallest could only get at most $\frac{1}{2^{k-1}}$ of the way around. Therefore, up to the time of election, the token carrying $u_{\min}$ uses more messages than all the others combined. Since $u_{\min}$ uses exactly n messages, the total number of messages sent, up to the time of election, is less than $2n$.

But also, note that by the time $u_{\min}$ gets all the way around the ring, all nodes know about this value, and so will refuse to send out any other tokens. It follows that $2n$ is an upper bound on the total number of messages that are *ever* sent by the algorithm (including the time after the *leader* output).

The time complexity, as mentioned above, is $n \cdot 2^{u_{\min}}$, since each node delays the token carrying UID $u_{\min}$ for $2^{u_{\min}}$ time units.

Variable start times. Unlike the *LCR* and *HS* algorithms, the *VariableSpeeds* algorithms cannot be used "as is" in the version of the synchronous model with variable start times. However, a modification of the algorithm works:

Modified *VariableSpeeds* algorithm:

Define a process to be a *starter* if it receives a *wakeup* message strictly before (i.e., at an earlier round than) receiving any ordinary (non-*null*) messages.

Each starter i initiates a token to travel around the ring, carrying its UID u_i; non-starters never initiate tokens. Initially, this token travels "fast," at the rate of one transmission per round, getting passed along by all the non-starters that are awakened by the arrival of the token, just until it first arrives at a starter. (This could be a different starter, or i itself.) After the

token arrives at a starter, the token continues its journey, but from now on at the "slow" rate of one transmission every 2^{u_i} rounds.

Meanwhile, each process keeps track of the smallest starter's UID that it has seen and discards any token carrying an identifier that is larger than this smallest one. If a token returns to its originator, the originator is elected.

The modified *VariableSpeeds* algorithm ensures that the starter process with the minimum UID is elected. Let $i_{\text{min-start}}$ denote this process.

Complexity analysis. We count the messages in three classes.

1. The messages involved in the initial fast transmission of tokens. There are just n of these.

2. The messages involved in the slow transmission of tokens, up to the time when $i_{\text{min-start}}$'s token first reaches a starter. This takes at most n rounds from when the first process awakens. During this time, a token carrying UID v could use at most $\frac{n}{2^v}$ messages, for a total of at most $\sum_{v=1}^{n} \frac{n}{2^v} < n$ messages.

3. The messages involved in the slow transmission of tokens, after the time when $i_{\text{min-start}}$'s token first reaches a starter. This analysis is similar to that for the basic *VariableSpeeds* algorithm. By the time the winning token gets all the way around the ring, the kth smallest starter's identifier could only get at most $\frac{1}{2^{k-1}}$ of the way around. Therefore, the total number of messages sent, up to the time of election, is less than $2n$. But by the time the winning token gets all the way around the ring, all nodes know about its value, and so will refuse to send out any other tokens; thus, $2n$ is an upper bound on the number of messages in this class.

Thus, the total communication complexity is at most $4n$.

The time complexity is $n + n \cdot 2^{u_{\text{min-start}}}$.

3.6 Lower Bound for Comparison-Based Algorithms

So far, we have presented several algorithms for leader election on a synchronous ring. The *LCR* and *HS* algorithms are comparison based, and the latter achieves a communication complexity bound of $O(n \log n)$ messages and a time bound of $O(n)$. The *TimeSlice* and *VariableSpeeds* algorithms, on the other hand, are not comparison based, and use $O(n)$ messages, but have a huge running time. In

this section, we show a lower bound of $\Omega(n \log n)$ messages for comparison-based algorithms. This lower bound holds even if we assume that communication is bidirectional and the ring size n is known to the processes. In the next section, we show a similar lower bound of $\Omega(n \log n)$ messages for non-comparison-based algorithms with bounded time complexity.

The result of this section is based on the difficulty of *breaking symmetry*. Recall the impossibility result in Theorem 3.1, which says that, because of symmetry, it is impossible to elect a leader in the absence of distinguishing information such as UIDs. The main idea in the following argument is that a certain amount of symmetry can arise even in the presence of UIDs. In this case, the UIDs allow symmetry to be broken, but it might require a large amount of communication to do so.

Recall that we are assuming throughout this chapter that the processes in the ring are all identical except for their UIDs. Thus, the start states of the processes are identical except for designated components that contain the process UID. In general, we have not imposed any constraints on how the message-generation and transition functions can use the UID information.

We assume for the rest of this chapter (this section and the next) that there is only one start state containing each UID. (As in the proof of Theorem 3.1, this assumption does not cause any loss of generality.) The advantage of this assumption is that it implies that the system (with a fixed assignment of UIDs) has exactly one execution.

A comparison-based algorithm obeys certain additional constraints, expressed by the following slightly informal definition. A UID-based ring algorithm is *comparison based* if the only ways that the processes manipulate the UIDs are by copying them, by sending and receiving them in messages, and by comparing them for $\{<, >, =\}$.

This definition allows a process, for example, to store any of the various UIDs that it has encountered and to send them out in messages, possibly combined with other information. A process can also compare the stored UIDs and use the results of these comparisons to make choices in the message-generation and state-transition functions. These choices could involve, for example, whether or not to send a message to each of its neighbors, whether or not to elect itself the leader, whether or not to keep the stored UIDs, and so on. The important fact is that all of the activity of a process depends only on the relative ranks of the UIDs it has encountered, rather than on their particular values.

The following formal notion is used to describe the kind of symmetry that can exist, even with UIDs. Let $U = (u_1, u_2, \ldots, u_k)$ and $V = (v_1, v_2, \ldots, v_k)$ be two sequences of UIDs, both of the same length k. We say that U is *order equivalent* to V if, for all $i, j, 1 \leq i, j \leq k$, we have $u_i \leq u_j$ if and only if $v_i \leq v_j$.

Example 3.6.1 Order equivalence

The sequences $(5,3,7,0)$, $(4,2,6,1)$, and $(5,3,6,1)$ are all order equivalent if the UID set is the natural numbers with the usual ordering.

Notice that two sequences of UIDs are order equivalent if and only if the corresponding sequences of relative ranks of their UIDs are identical. Two technical definitions follow. A round of an execution is said to be *active* if at least one (non-null) message is sent in it. The *k-neighborhood* of process i in ring R of size n, where $0 \leq k < \lfloor n/2 \rfloor$, is defined to consist of the $2k+1$ processes $i-k, \ldots, i+k$, that is, those that are within distance at most k from process i (including i itself).

Finally, we need a definition of what it means for process states to be the same, except for the particular choices of UIDs they contain. We say that two process states s and t *correspond* with respect to sequences $U = (u_1, u_2, \ldots, u_k)$ and $V = (v_1, v_2, \ldots, v_k)$ of UIDs provided that the following hold: all the UIDs in s are chosen from U, all the UIDs in t are chosen from V, and t is identical to s except that each occurrence of u_i in s is replaced by an occurrence of v_i in t, for all i, $1 \leq i \leq k$. *Corresponding messages* are defined analogously.

We can now prove the key lemma for our lower bound, Lemma 3.5. It says that processes that have order-equivalent k-neighborhoods behave in essentially the same way, until information has had a chance to propagate to the processes from outside the k-neighborhoods.

Lemma 3.5 *Let A be a comparison-based algorithm executing in a ring R of size n and let k be an integer, $0 \leq k < \lfloor n/2 \rfloor$. Let i and j be two processes in A that have order-equivalent sequences of UIDs in their k-neighborhoods. Then, at any point after at most k active rounds, processes i and j are in corresponding states, with respect to the UID sequences in their k-neighborhoods.*

Example 3.6.2 Corresponding states

Suppose that the sequence of UIDs in process i's 3-neighborhood is $(1,6,3,8,4,10,7)$ (where process i's UID is 8), and the sequence in process j's 3-neighborhood is $(4,10,7,12,9,13,11)$ (where process j's UID is 12). Since these two sequences are order equivalent, Lemma 3.5 implies that processes i and j remain in corresponding states with respect to their 3-neighborhoods, as long as no more than three active rounds have occurred. Roughly speaking, the reason this is so is that if there are only three active rounds, there has not been any opportunity for information from outside the order-equivalent 3-neighborhoods to reach i and j.

Proof (of Lemma 3.5). Without loss of generality, we may assume that $i \neq j$. We proceed by induction on the number r of rounds that have been performed in the execution. For each r, we prove the lemma for all k.

Basis: $r = 0$. By the definition of a comparison-based algorithm, the initial states of i and j are identical except for their own UIDs, and hence they are in corresponding initial states, with respect to their k-neighborhoods (for any k).

Inductive step: Assume that the lemma holds for all $r' < r$. Fix k such that i and j have order-equivalent k-neighborhoods, and suppose that the first r rounds include at most k active rounds.

If neither i nor j receives a message at round r, then by induction (for $r-1$ and k), i and j are in corresponding states just after $r-1$ rounds, with respect to their k-neighborhoods. Since i and j have no new input, they make corresponding transitions and end up in corresponding states after round r.

So assume that either i or j receives a message at round r. Then, round r is active, so the first $r-1$ rounds include at most $k-1$ active rounds. Note that i and j have order-equivalent $(k-1)$-neighborhoods, and likewise for $i-1$ and $j-1$ and for $i+1$ and $j+1$. Therefore, by induction (for $r-1$ and $k-1$), we have that i and j are in corresponding states after $r-1$ rounds, with respect to their $(k-1)$-neighborhoods, and similarly for $i-1$ and $j-1$, and for $i+1$ and $j+1$.

We proceed by case analysis.

1. At round r, neither $i-1$ nor $i+1$ sends a message to i.

   Then, since $i-1$ and $j-1$ are in corresponding states after $r-1$ rounds, and likewise for $i+1$ and $j+1$, we have that neither $j-1$ nor $j+1$ sends a message to j at round r. But this contradicts the assumption that either i or j receives a message at round r.

2. At round r, $i-1$ sends a message to i but $i+1$ does not.

   Then, since $i-1$ and $j-1$ are in corresponding states after $r-1$ rounds, $j-1$ also sends a message to j at round r, and that message corresponds to the message sent by $i-1$ to i, with respect to the $(k-1)$-neighborhoods of $i-1$ and $j-1$, and hence with respect to the k-neighborhoods of i and j. For similar reasons, $j+1$ sends no message to j at round r. Since i and j are in corresponding states after round $r-1$, and receive corresponding messages, they remain in corresponding states, this time with respect to their k-neighborhoods.

3. At round r, $i+1$ sends a message to i but $i-1$ does not.

   Analogous to the previous case.

4. At round r, both $i-1$ and $i+1$ send messages to i.

A similar argument. □

Lemma 3.5 tells us that many active rounds are necessary to break symmetry if there are large order-equivalent neighborhoods. We now define particular rings with the special property that they have many order-equivalent neighborhoods of various sizes. Let $c, 0 \leq c \leq 1$, be a constant, and let R be a ring of size n. Then R is said to be *c-symmetric* if for every l, $\sqrt{n} \leq l \leq n$, and for every segment S of R of length l, there are at least $\lfloor \frac{cn}{l} \rfloor$ segments in R that are order equivalent to S (counting S itself).[1]

If n is a power of 2, then it is easy to construct a ring that is $\frac{1}{2}$-symmetric. Specifically, we define the *bit-reversal ring* of size n as follows. Suppose that $n = 2^k$. Then we assign to each process i the integer in the range $[0, n-1]$ whose k-bit binary representation is the reverse of the k-bit binary representation of i (we use 0^k as the k-bit binary representation of n, identifying n with 0).

Example 3.6.3 Bit-reversal ring

For $n = 8$, we have $k = 3$, and the assignment is as in Figure 3.3.

Lemma 3.6 *Any bit-reversal ring is $\frac{1}{2}$-symmetric.*

Proof. Left as an exercise.[2] □

For values of n that are not powers of 2, there also always exist c-symmetric rings, though the general case requires a smaller constant c.

Theorem 3.7 *There exists a constant c such that, for all $n \in \mathbb{N}^+$, there is a c-symmetric ring of size n.*

The proof of Theorem 3.7 involves a fairly complicated recursive construction.[3] It is not possible to produce the needed ring simply, say by starting with the bit-reversal ring for the next smaller power of 2 and just adding some extra processes; these extra processes would destroy the symmetry.

So we can assume, for any n, that we have a c-symmetric ring R of size n. The following lemma states that if such a ring elects a leader, then it must have many active rounds.

[1] Try to ignore the square root lower bound condition—it is just a technicality.

[2] Note that for the bit-reversal ring, there is no need for the square root lower bound condition.

[3] This is where the square root lower bound condition arises.

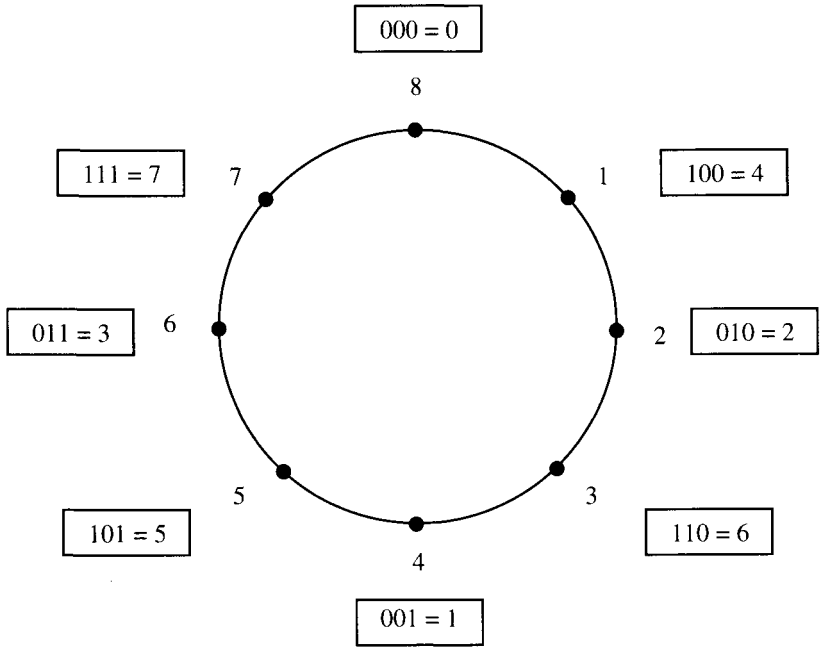

Figure 3.3: Bit-reversal ring of size 8.

Lemma 3.8 *Let A be a comparison-based algorithm executing in a c-symmetric ring of size n, and suppose that A elects a leader. Suppose that k is an integer such that $\sqrt{n} \leq 2k+1$ and $\left\lfloor \frac{cn}{2k+1} \right\rfloor \geq 2$. Then A has more than k active rounds.*

Proof. We proceed by contradiction. Suppose that A elects a leader, say process i, in at most k active rounds. Let S be the k-neighborhood of i; S is a segment of length $2k+1$. Since the ring is c-symmetric, there must be at least $\left\lfloor \frac{cn}{2k+1} \right\rfloor \geq 2$ segments in the ring that are order equivalent to S, counting S itself. Thus, there is at least one other segment that is order equivalent to S; let j be the midpoint of that segment. Now, by Lemma 3.5, i and j remain in equivalent states throughout the execution, up to the election point. We conclude that j also gets elected, a contradiction. □

Now we can prove the lower bound.

Theorem 3.9 *Let A be a comparison-based algorithm that elects a leader in rings of size n. Then there is an execution of A in which $\Omega(n \log n)$ messages are sent by the time the leader is elected.*[4]

[4]The $\Omega(n \log n)$ expression hides a fixed constant, independent of n.

Proof. Fix c to be the constant whose existence is asserted by Theorem 3.7, and use that theorem to obtain a c-symmetric ring R of size n. We consider executions of the algorithm in ring R.

Define $k = \left\lfloor \frac{cn-2}{4} \right\rfloor$. Then $\sqrt{n} \leq 2k+1$ (provided n is sufficiently large), and $\left\lfloor \frac{cn}{2k+1} \right\rfloor \geq 2$. It follows by Lemma 3.8 that there are more than k active rounds, that is, that there are at least $k+1$ active rounds.

Consider the rth active round, where $\sqrt{n}+1 \leq r \leq k+1$. Since the round is active, there is some process i that sends a message in round r. Let S be the $(r-1)$-neighborhood of i. Since R is c-symmetric, there are at least $\left\lfloor \frac{cn}{2r-1} \right\rfloor$ segments in R that are equivalent to S. By Lemma 3.5, at the point just before the rth active round, the midpoints of all these segments are in corresponding states, so they all send messages.

Now let $r_1 = \lceil \sqrt{n} \rceil + 1$ and $r_2 = k+1 = \left\lfloor \frac{cn-2}{4} \right\rfloor + 1$. The argument above implies that the total number of messages is at least

$$\sum_{r=r_1}^{r_2} \left\lfloor \frac{cn}{2r-1} \right\rfloor \geq \sum_{r=r_1}^{r_2} \frac{cn}{2r-1} - r_2.$$

The second term is $O(n)$, so it suffices to show that the first term is $\Omega(n \log n)$. We have

$$\begin{aligned} \sum_{r=r_1}^{r_2} \frac{cn}{2r-1} &= \Omega\left(n \sum_{r=r_1}^{r_2} \frac{1}{r}\right) \\ &= \Omega\left(n \left(\ln r_2 - \ln r_1\right)\right) \end{aligned}$$

by an integral approximation of the sum,

$$\begin{aligned} &= \Omega\left(n \left(\ln\left(\left\lfloor \frac{cn-2}{4} \right\rfloor + 1\right) - \ln(\lceil \sqrt{n} \rceil + 1)\right)\right) \\ &= \Omega(n \log n). \end{aligned}$$

This is as needed. □

3.7 Lower Bound for Non-Comparison-Based Algorithms*

Can we describe any lower bounds on the number of messages for the case of non-comparison-based algorithms? Although the $\Omega(n \log n)$ barrier can be broken in this case, it is possible to show that this can only happen at the cost of large time complexity. For example, suppose that the time until leader election is bounded

by t. Then, if the total number of UIDs in the space of identifiers is sufficiently large—say, greater than some particular fast-growing function $f(n,t)$—then there is a subset U of the identifiers on which it is possible to show that the algorithm behaves "like a comparison-based algorithm," at least through t rounds. This implies that the lower bound for comparison carries over to the time-bounded algorithm using identifiers in U.

We give somewhat more detail, but our presentation is still just a sketch. We will define the fast-growing function $f(n,t)$ using *Ramsey's Theorem*, which is a kind of generalized Pigeonhole Principle. In the statement of the theorem, an n-subset is just a subset with n elements, and a coloring just assigns a color to each set.

Theorem 3.10 (Ramsey's Theorem) *For all integers n, m, and c, there exists an integer $g(n,m,c)$ with the following property. For every set S of size at least $g(n,m,c)$, and any coloring of the n-subsets of S with at most c colors, there is some subset C of S of size m that has all of its n-subsets colored the same color.*

We begin by putting each algorithm into a *normal form*, in which each state simply records, in LISP S-expression format, the initial UID plus all the messages ever received, and each non-*null* message contains the complete state of its sender. Certain of these S-expressions are then designated as *election* states, in which the process is identified as having been elected as the leader. If the original algorithm is a correct leader-election algorithm, then the new one (with the modified output convention) is also, and the communication complexity is the same.

Our lower bound theorem is as follows.

Theorem 3.11 *For all integers n and t, there exists an integer $f(n,t)$ with the following property. Let A be any (not necessarily comparison-based) algorithm that elects a leader in rings of size n within time t and uses a UID space of size at least $f(n,t)$. Then there is an execution of A in which $\Omega(n \log n)$ messages are sent by the time the leader is elected.*

Proof Sketch. Fix n and t. Without loss of generality, we only consider algorithms in normal form. Since the algorithms involve only n processes and proceed for only t rounds, all the S-expressions that arise have at most n distinct arguments and at most t parenthesis depth.

Now for each algorithm A, we define an equivalence relation $\equiv_A$ on n-sets (i.e., sets of size n) of UIDs; roughly speaking, two n-sets will be said to be equivalent if they give rise to the same behavior for algorithm A. More precisely, if V and V' are two n-sets of UIDs, then we say that $V \equiv_A V'$ if, for every

S-expression of depth at most t over V, the corresponding S-expression over V' (generated by replacing each element of V with the same rank element within V') give rise to the same decisions, in algorithm A, about whether to send a message in each direction and about whether or not the process is elected as leader.

Because the S-expressions in the definition of the equivalence relation have at most n arguments and at most t depth, there are only finitely many $\equiv_A$ equivalence classes; in fact, there is an upper bound on the number of classes that does not depend on the algorithm A, but only on n and t. Let $c(n,t)$ be such an upper bound.

Now fix algorithm A. We describe a way of coloring n-sets of UIDs, so we can apply Ramsey's Theorem. Namely, we just associate a color with each $\equiv_A$ equivalence class of n-sets, and color all the n-sets in that class by that color.

Now define $f(n,t) = g(n, 2n, c(n,t))$, where g is the function in Theorem 3.10, and consider any UID space containing at least $f(n,t)$ identifiers. Then, Theorem 3.10 implies the existence of a subset C of the UID space, containing at least $2n$ elements, such that all n-subsets of C are colored the same color. Take U to be the set consisting of the n smallest elements of C.

Then we claim that the algorithm behaves exactly like a comparison algorithm through t rounds, when UIDs are chosen from U. That is, every decision made by any process, about whether to send a message in either direction or about whether the process is a leader, depends only on the relative order of the arguments contained in the current state. To see why this is so, fix any two subsets W and W' of U, of the same size—say m. Suppose that S is an S-expression of depth at most t with UIDs chosen from W, and S' is the corresponding S-expression over W' (generated by replacing each element of W with the same rank element within W'). Then W and W' can be extended to sets V and V', each of size exactly n, by including the $n-m$ largest elements of C. Since V and V' are colored the same color, the two S-expressions give rise to the same decisions about whether to send a message in each direction and about whether or not the process is elected as leader.

Since the algorithm behaves exactly like a comparison algorithm through t rounds, when UIDs are chosen from U, Theorem 3.9 yields the lower bound. □

3.8 Bibliographic Notes

The impossibility result of Section 3.2 seems to be a part of the ancient folklore of this area; one version of this result, for a different model, appears in a paper by Angluin [13]. The *LCR* algorithm is derived from one developed by Le Lann