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
3. LEADER ELECTION IN A SYNCHRONOUS RING

![Diagram of a ring of processes](image)

**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
3.2. IMPOSSIBILITY RESULT FOR IDENTICAL PROCESSES

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:

- $v$, 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 $msgs_i$ 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
```
3.3. A BASIC ALGORITHM

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 \texttt{trans;} 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 \texttt{leader} output. Let \( i_{\text{max}} \) denote the index of the process with the maximum UID, and let \( u_{\text{max}} \) denote its UID. It is enough to show that (1) process \( i_{\text{max}} \) outputs \texttt{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 \( i_{\text{max}} \) outputs \texttt{leader} by the end of round \( n \).

**Proof.** Note that \( u_{\text{max}} \) is the initial value of variable \( u_{i_{\text{max}}} \), the variable \( u \) at process \( i_{\text{max}} \), 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 \( i_{\text{max}} \) has the largest \( u \) value (by definition of \( i_{\text{max}} \)). By the code, it suffices to show the following invariant assertion:

**Assertion 3.3.1** After \( n \) rounds, \( \text{status}_{i_{\text{max}}} = \text{leader} \).

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, \( \text{send}_{i_{\text{max}} + r} = u_{\text{max}} \).

(Recall that addition is modulo \( n \).) This assertion says that the maximum value appears in the \texttt{send} component at the position in the ring at distance \( r \) from \( i_{\text{max}} \).
It is straightforward to prove Assertion 3.3.2 by induction on \( r \). For \( r = 0 \), the initialization says that \( \text{send}_\max = u_{\max} \) 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 \textit{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 \textit{status} to \textit{leader}.

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

**Proof.** It is enough to show that all other processes always have \textit{status} = \textit{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 \textit{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 \( \text{send}_j \neq u_i \).

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 \textit{leader}.

**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 \textit{report} message to be sent around the ring. Any process that receives the \textit{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 \textit{non-leader}. Furthermore, by attaching the leader's index to the \textit{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
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, … . 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 traveling 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:

1. send the current value of $send^+$ to process $i + 1$
2. send the current value of $send^-$ to process $i - 1$

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

```plaintext
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
```

```
$send^+ := (u, in, 1)$
if the message from $i + 1$ is $(v, in, 1)$ and $v \neq u$ then
$send^- := (u, in, 1)$
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.