Chapter 5

Leader election

See [AW04, Chapter 3] or [Lyn96, Chapter 3] for details.

The basic idea of leader election is that we want a single process to declare itself leader and the others to declare themselves non-leaders. The non-leaders may or may not learn the identity of the leader as part of the protocol; if not, we can always add an extra phase where the leader broadcasts its identity to the others. Traditionally, leader election has been used as a way to study the effects of symmetry, and many leader election algorithms are designed for networks in the form of a ring.

A classic result of Angluin [Ang80] shows that leader election in a ring is impossible if the processes do not start with distinct identities. The proof is that if everybody is in the same state at every step, they all put on the crown at the same time. We discuss this result in more detail in §5.1.

With ordered identities, a simple algorithm due to Le Lann [LL77] and Chang and Roberts [CR79] solves the problem in $O(n)$ time with $O(n^2)$ messages: I send out my own id clockwise and forward any id bigger than mine. If I get my id back, I win. This works with a unidirectional ring, doesn't require synchrony, and never produces multiple leaders. See §5.2.1 for more details.

On a bidirectional ring we can get $O(n \log n)$ messages and $O(n)$ time with power-of-2 probing, using an algorithm of Hirschberg and Sinclair [HS80]. This is described in §5.2.2.

An evil trick: if we have synchronized starting, known n, and known id space, we can have process with id i wait until round $i \cdot n$ to start sending its id around, and have everybody else drop out when they receive it; this way only one process (the one with smallest id) ever starts a message and only n messages are sent [FL87]. But the running time can be pretty bad.

For general networks, we can apply the same basic strategy as in Le Lann-Chang-Roberts by having each process initiate a broadcast/convergecast algorithm that succeeds only if the initiator has the smallest id. This is described in more detail in §5.3.

Some additional algorithms for the asynchronous ring are given in §§5.2.3 and 5.2.4. Lower bounds are shown in §5.4.

5.1 Symmetry

A system exhibits **symmetry** if we can permute the nodes without changing the behavior of the system. More formally, we can define a symmetry as an **equivalence relation** on processes, where we have the additional properties that all processes in the same equivalence class run the same code; and whenever p is equivalent to p', each neighbor q of p is equivalent to the corresponding neighbor q' of p'.

An example of a network with a lot of symmetries would be an **anonymous ring**, which is a network in the form of a cycle (the ring part) in which every process runs the same code (the anonymous part). In this case all nodes are equivalent. If we have a line, then we might or might not have any non-trivial symmetries: if each node has a **sense of direction** that tells it which neighbor is to the left and which is to the right, then we can identify each node uniquely by its distance from the left edge. But if the nodes don't have a sense of direction, we can flip the line over and pair up nodes that map to each other.[1]

Symmetries are convenient for proving impossibility results, as observed by Angluin [Ang80]. The underlying theme is that without some mechanism for **symmetry breaking**, a message-passing system escape from a symmetric initial configuration. The following lemma holds for **deterministic** systems, basically those in which processes can't flip coins:

Lemma 5.1.1. *A symmetric deterministic message-passing system that starts in an initial configuration in which equivalent processes have the same state has a synchronous execution in which equivalent processes continue to have the same state.*

Proof. Easy induction on rounds: if in some round p and p' are equivalent and have the same state, and all their neighbors are equivalent and have the

[1]Typically, this does not mean that the nodes can't tell their neighbors apart. But it does mean that if we swap the labels for all the neighbors (corresponding to flipping the entire line from left to right), we get the same executions.

same state, then p and p' receive the same messages from their neighbors and can proceed to the same state (including outgoing messages) in the next round. $\square$

An immediate corollary is that you can't do leader election in an anonymous system with a symmetry that puts each node in a non-trivial equivalence class, because as soon as I stick my hand up to declare I'm the leader, so do all my equivalence-class buddies.

With **randomization**, Lemma 5.1.1 doesn't directly apply, since we can break symmetry by having my coin-flips come up differently from yours. It does show that we can't guarantee convergence to a single leader in any fixed amount of time (because otherwise we could just fix all the coin flips to get a deterministic algorithm). Depending on what the processes know about the size of the system, it may still be possible to show that a randomized algorithm necessarily fails in some cases.[2]

A more direct way to break symmetry is to assume that all processes have **identities**; now processes can break symmetry by just declaring that the one with the smaller or larger identity wins. This approach is taken in the algorithms in the following sections.

5.2 Leader election in rings

Here we'll describe some basic leader election algorithms for rings. Historically, rings were the first networks in which leader election was studied, because they are the simplest networks whose symmetry makes the problem difficult, and because of the connection to token-ring networks, a method for congestion control in local-area networks that is no longer used much.

5.2.1 The Le-Lann-Chang-Roberts algorithm

This is about the simplest leader election algorithm there is. It works in a **unidirectional ring**, where messages can only travel clockwise.[3] The algorithms works does not require synchrony, but we'll assume synchrony to make it easier to follow.

[2]Specifically, if the processes don't know the size of the ring, we can imagine a ring of size $2n$ in which the first n processes happen to get exactly the same coin-flips as the second n processes for long enough that two matching processes, one in each region, both think they have won the fight in a ring of size n and declare themself to be the leader.

[3]We'll see later in §5.2.3 that the distinction between unidirectional rings and bidirectional rings is not a big deal, but for now let's imagine that having a unidirectional ring is a serious hardship.

Formally, we'll let the state space for each process i consist of two variables: leader, initially 0, which is set to 1 if i decides it's a leader; and maxId, the largest id seen so far. We assume that i denotes i's position rather than its id, which we'll write as id_i. We will also treat all positions as values mod n, to simplify the arithmetic.

Code for the LCR algorithm is given in Algorithm 5.1.

```
initially do
    leader ← 0
    maxId ← $\mathsf{id}_i$
    send $\mathsf{id}_i$ to clockwise neighbor
upon receiving $j$ do
    if $j = \mathsf{id}_i$ then
        leader ← 1
    if $j >$ maxId then
        maxId ← $j$
        send $j$ to clockwise neighbor
```

Algorithm 5.1: LCR leader election

5.2.1.1 Proof of correctness for synchronous executions

By induction on the round number k. The induction hypothesis is that in round k, each process i's leader bit is 0, its maxId value is equal to the largest id in the range $(i-k)\dots i$, and that it sends id_{i-k} if and only if id_{i-k} is the largest id in the range $(i-k)\dots i$. The base case is that when $k=0$, $\mathsf{maxId} = \mathsf{id}_i$ is the largest id in $i\dots i$, and i sends id_i. For the induction step, observe that in round $k-1$, $i-1$ sends $\mathsf{id}_{(i-1)-(k-1)} = \mathsf{id}_{i-k}$ if and only if it is the largest in the range $(i-k)\dots(i-1)$, and that i adopts it as the new value of maxId and sends it just in case it is larger than the previous largest value in $(i-k+1)\dots(i-1)$, i.e., if it is the largest value in $(i-k)\dots i$.

Finally, in round $n-1$, $i-1$ sends $\mathsf{id}_{i-N} = \mathsf{id}_i$ if and only if i is the largest id in $(i-n+1)\dots i$, the whole state space. So i receives id_i and sets $\mathsf{leader}_i = 1$ if and only if it has the maximum id.

5.2.1.2 Performance

It's immediate from the correctness proof that the protocols terminates after exactly n rounds.

To count message traffic, observe that each process sends at most 1 message per round, for a total of $O(n^2)$ messages. This is a tight bound since if the ids are in decreasing order $n, n-1, n-2, \ldots 1$, then no messages get eaten until they hit n.

5.2.2 The Hirschberg-Sinclair algorithm

This algorithm improves on Le-Lann-Chang-Roberts by reducing the message complexity. The idea is that instead of having each process send a message all the way around a ring, each process will first probe locally to see if it has the largest id within a short distance. If it wins among its immediate neighbors, it doubles the size of the neighborhood it checks, and continues as long as it has a winning id. This means that most nodes drop out quickly, giving a total message complexity of $O(n \log n)$. The running time is a constant factor worse than LCR, but still $O(n)$.

To specify the protocol, it may help to think of messages as mobile agents and the state of each process as being of the form (local-state, {agents I'm carrying}). Then the sending rule for a process becomes *ship any agents in whatever direction they want to go* and the transition rule is *accept any incoming agents and update their state in terms of their own internal transition rules.* An agent state for LCR will be something like (original-sender, direction, hop-count, max-seen) where direction is R or L depending on which way the agent is going, hop-count is initially 2^k when the agent is sent and drops by 1 each time the agent moves, and max-seen is the biggest id of any node the agent has visited. An agent turns around (switches direction) when hop-count reaches 0.

To prove this works, we can mostly ignore the early phases (though we have to show that the max-id node doesn't drop out early, which is not too hard). The last phase involves any surviving node probing all the way around the ring, so it will declare itself leader only when it receives its own agent from the left. That exactly one node does so is immediate from the same argument for LCR.

Complexity analysis is mildly painful but basically comes down to the fact that any node that sends a message 2^k hops had to be a winner at phase 2^{k-1}, which means that it is the largest of some group of 2^{k-1} ids. Thus the 2^k-hop senders are spaced at least 2^{k-1} away from each other and there are at most $n/2^{k-1}$ of them. Summing up over all $\lceil \lg n \rceil$ phases, we get $\sum_{k=0}^{\lceil \lg n \rceil} 2^k n/2^{k-1} = O(n \log n)$ messages and $\sum_{k=0}^{\lceil \lg n \rceil} 2^k = O(n)$ time.

5.2.3 Peterson's algorithm for the unidirectional ring

This algorithm is due to Peterson [Pet82] and assumes an asynchronous, unidirectional ring. It gets $O(n \log n)$ message complexity in all executions.

The basic idea (2-way communication version): Start with n candidate leaders. In each of at most $\lg n$ asynchronous phases, each candidate probes its nearest neighbors to the left and right; if its ID is larger than the IDs of both neighbors, it survives to the next phase. Non-candidates act as relays passing messages between candidates. As in Hirschberg and Sinclair (§5.2.2), the probing operations in each phase take $O(n)$ messages, and at least half of the candidates drop out in each phase. The last surviving candidate wins when it finds that it's its own neighbor.

To make this work in a 1-way ring, we have to simulate 2-way communication by moving the candidates clockwise around the ring to catch up with their unsendable counterclockwise messages. Peterson's algorithm does this with a two-hop approach that is inspired by the 2-way case above; in each phase k, a candidate effectively moves two positions to the right, allowing it to look at the ids of three phase-k candidates before deciding to continue in phase $k + 1$ or not. Here is a very high-level description; it assumes that we can buffer and ignore incoming messages from the later phases until we get to the right phase, and that we can execute sends immediately upon receiving messages. Doing this formally in terms of the model of §2.1 means that we have to build explicit internal buffers into our processes, which we can easily do but won't do here (see [Lyn96, pp. 483–484] for the right way to do this.)

We can use a similar trick to transform any bidirectional-ring algorithm into a unidirectional-ring algorithm: alternative between phases where we send a message right, then send a virtual process right to pick up any left-going messages deposited for us. The problem with this trick is that it requires two messages per process per phase, which gives us a total message complexity of $O(n^2)$ if we start with an $O(n)$-time algorithm. Peterson's algorithm avoids this by only propagating the surviving candidates.

Pseudocode for Peterson's algorithm is given in Algorithm 5.2.

Note: the phase arguments in the probe messages are useless if one has FIFO channels, which is why [Lyn96] doesn't use them. Note also that the algorithm does not elect the process with the highest ID, but the process that is carrying the sole surviving candidate in the last phase.

Proof of correctness is essentially the same as for the 2-way algorithm. For any pair of adjacent candidates, at most one of their current IDs survives to the next phase. So we get a sole survivor after $\lg n$ phases. Each process

```
procedure candidate()
    phase ← 0
    current ← pid
    while true do
        send probe(phase, current)
        wait for probe(phase, $x$)
        $\mathsf{id}_2$ ← $x$
        send probe(phase, current)
        wait for probe(phase, $x$)
        $\mathsf{id}_3$ ← $x$
        if $\mathsf{id}_2$ = current then
            I am the leader!
            return
        else if $\mathsf{id}_2$ > current and $\mathsf{id}_2$ > $\mathsf{id}_3$ do
            current ← $\mathsf{id}_2$
            phase ← phase + 1
        else
            switch to relay()

procedure relay()
    upon receiving probe($p, i$) do
        send probe($p, i$)
```

Algorithm 5.2: Peterson's leader-election algorithm

sends or relays at most 2 messages per phases, so we get at most $2n \lg n$ total messages.

5.2.4 A simple randomized $O(n \log n)$-message algorithm

An alternative to running a more sophisticated algorithm is to reduce the average cost of LCR using randomization. The presentation here follows the average-case analysis done by Chang and Roberts [CR79].

Run LCR where each id is constructed by prepending a long random bit-string to the real id. This gives uniqueness (since the real id's act as tie-breakers) and something very close to a random permutation on the constructed id's. When we have unique random id's, a simple argument shows that the i-th largest id only propagates an expected n/i hops, giving a total of $O(nH_n) = O(n \log n)$ hops.[4] Unique random id's occur with high probability provided the range of the random sequence is $\gg n^2$.

The downside of this algorithm compared to Peterson's is that knowledge of n is required to pick random id's from a large enough range. It also has higher bit complexity since Peterson's algorithm is sending only IDs (in the official version) without any random padding.

5.3 Leader election in general networks

For general networks, a simple approach is to have each node initiate a breadth-first-search and convergecast, with nodes refusing to participate in the protocol for any initiator with a lower id. It follows that only the node with the maximum id can finish its protocol; this node becomes the leader. If messages from parallel broadcasts are combined, it's possible to keep the message complexity of this algorithm down to $O(DE)$.

More sophisticated algorithms reduce the message complexity by coalescing local neighborhoods similar to what happens in the Hirschberg-Sinclair and Peterson algorithms. A noteworthy example is an $O(n \log n)$ message-complexity algorithm of Afek and Gafni [AG91], who also show an $\Omega(n \log n)$ lower bound on message complexity for any synchronous algorithm in a complete network.

[4] Alternatively, we could consider the **average-case complexity** of the algorithm when we assume all $n!$ orderings of the ids are equally likely; this also gives $O(n \log n)$ expected message complexity [CR79].

5.4 Lower bounds

Here we present two classic $\Omega(\log n)$ lower bounds on message complexity for leader election in the ring. The first, due to Burns [Bur80], assumes that the system is asynchronous and that the algorithm is **uniform**: it does not depend on the size of the ring. The second, due to Frederickson and Lynch [FL87], allows a synchronous system and relaxes the uniformity assumption, but requires that the algorithm can't do anything to ids but copy and compare them.

5.4.1 Lower bound on asynchronous message complexity

Here we describe a lower bound for uniform asynchronous leader election in the ring. The description here is based on [AW04, §3.3.3]; a slightly different presentation can also be found in [Lyn96, §15.1.4]. The original result is due to Burns [Bur80]. We assume the system is deterministic.

The basic idea is to construct a bad execution in which n processes send lots of messages recursively, by first constructing two bad $(n/2)$-process executions and pasting them together in a way that generates many extra messages. If the pasting step produces $\Theta(n)$ additional messages, we get a recurrence $T(n) \geq 2T(n/2) + \Theta(n)$ for the total message traffic, which has solution $T(n) = \Omega(n \log n)$.

We'll assume that all processes are trying to learn the identity of the process with the smallest id. This is a slightly stronger problem that mere leader election, but it can be solved with at most an additional $2n$ messages once we actually elect a leader. So if we get a lower bound of $f(n)$ messages on this problem, we immediately get a lower bound of $f(n) - 2n$ on leader election.

To construct the bad execution, we consider "open executions" on rings of size n where no message is delivered across some edge (these will be partial executions, because otherwise the guarantee of eventual delivery kicks in). Because no message is delivered across this edge, the processes can't tell if there is really a single edge there or some enormous unexplored fragment of a much larger ring. Our induction hypothesis will show that a line of $n/2$ processes can be made to send at least $T(n/2)$ messages in an open execution (before seeing any messages across the open edge); we'll then show that a linear number of additional messages can be generated by pasting two such executions together end-to-end, while still getting an open execution with n processes.

In the base case, we let $n = 1$. Somebody has to send a message eventually, giving $T(2) \geq 1$.

For larger n, suppose that we have two open executions on $n/2$ processes that each send at least $T(n/2)$ messages. Break the open edges in both executions and paste the resulting lines together to get a ring of size n; similarly paste the schedules σ_1 and σ_2 of the two executions together to get a combined schedule $\sigma_1\sigma_2$ with at least $2T(n/2)$ messages. Note that in the combined schedule no messages are passed between the two sides, so the processes continue to behave as they did in their separate executions.

Let e and e' be the edges we used to past together the two rings. Extend $\sigma_1\sigma_2$ by the longest possible suffix σ_3 in which no messages are delivered across e and e'. Since σ_3 is as long as possible, after $\sigma_1\sigma_2\sigma_3$, there are no messages waiting to be delivered across any edge except e and e' and all processes are **quiescent**—they will send no additional messages until they receive one.

We now consider some suffix σ_4 such causes the protocol to finish. While executing σ_4, construct two sets of processes S and S' by the following rules:

1. If a process is not yet in S or S' and receives a message delivered across e, put it in S; similarly if it receives a message delivered across e', put it in S'.

2. If a process is not yet in S or S' and receives a message that was sent by a process in S, put it in S; similarly for S'.

triggered by a delivery across e

Observe that $S \cup S'$ includes every process on the half of the ring with the larger minimum id, because any such process that doesn't receive a message in σ_4 doesn't learn the global minimum. So $|S \cup S'| \geq n/2$ and thus $\min(|S|, |S'|) \geq n/4$.

Assume without loss of generality that it is $|S|$ that is at least $n/4$. Except for the two processes incident to e, every process that is added to S is added in response to a message sent in σ_4. So there are at least $n/4 - 2$ such messages. We can also argue that all of these messages are sent in the subschedule τ of σ_4 that contains only messages that do not depend on messages delivered across e'. It follows that $\sigma_1\sigma_2\sigma_3\tau$ is an open execution on n processes with at least $2T(n/2) + n/4 - 2$ sent messages. This gives $T(n) \geq 2T(n/2) + n/4 - 2 = 2T(n/2) + \Omega(n)$ as claimed.

5.4.2 Lower bound for comparison-based algorithms

Here we give an $\Omega(n \log n)$ lower bound on messages for synchronous-start comparison-based algorithms in bidirectional synchronous rings. For full details see [Lyn96, §3.6], [AW04, §3.4.2], or the original JACM paper by Frederickson and Lynch [FL87].

Basic ideas:

- Two fragments $i \dots i+k$ and $j \dots j+k$ of a ring are **order-equivalent** provided $\mathsf{id}_{i+a} > \mathsf{id}_{i+b}$ if and only if $\mathsf{id}_{j+a} > \mathsf{id}_{j+b}$ for $b = 0 \dots k$.

- An algorithm is **comparison-based** if it can't do anything to IDs but copy them and test for $<$. The state of such an algorithm is modeled by some non-ID state together with a big bag of IDs, messages have a pile of IDs attached to them, etc. Two states/messages are equivalent under some mapping of IDs if you can translate the first to the second by running all IDs through the mapping.

  An equivalent version uses an explicit equivalence relation between processes. Let executions of p_1 and p_2 be **similar** if both processes send messages in the same direction(s) in the same rounds and both processes declare themselves leader (or not) at the same round. Then an algorithm is comparison-based based if order-equivalent rings yield similar executions for corresponding processes. This can be turned into the explicit-copying-IDs model by replacing the original protocol with a **full-information protocol** in which each message is replaced by the ID and a complete history of the sending process (including all messages it has every received).

- Define an **active round** as a round in which at least 1 message is sent. Claim: actions of i after k active rounds depends up to an order-equivalent mapping of ids only on the order-equivalence class of ids in $i-k \dots i+k$ (the k**-neighborhood** of i). Proof: by induction on k. Suppose i and j have order-equivalent $(k-1)$-neighborhoods; then after $k-1$ active rounds they have equivalent states by the induction hypothesis. In inactive rounds, i and j both receive no messages and update their states in the same way. In active rounds, i and j receive order-equivalent messages and update their states in an order-equivalent way.

- If we have an order of ids with a lot of order-equivalent k-neighborhoods, then after k active rounds if one process sends a message, so do a lot of other ones.

Now we just need to build a ring with a lot of order-equivalent neighborhoods. For n a power of 2 we can use the bit-reversal ring, e.g., id sequence

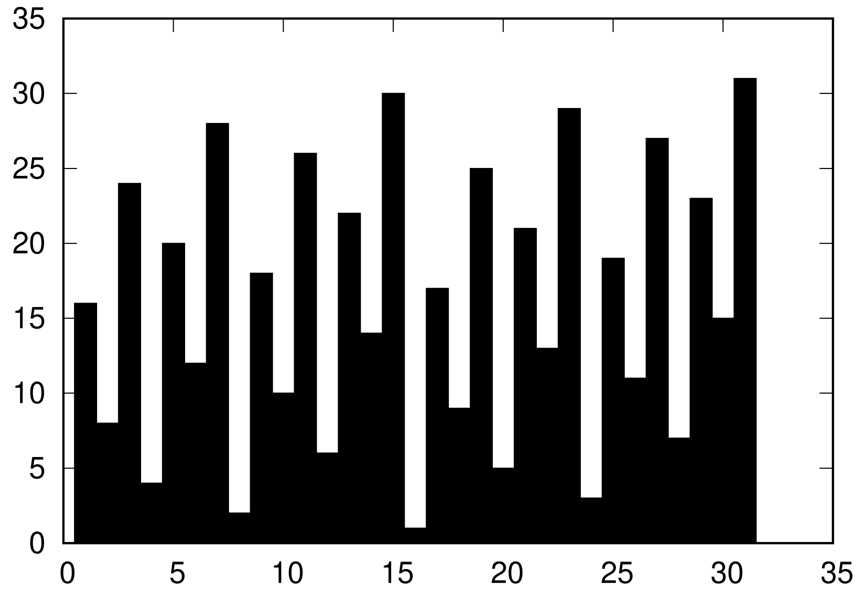

Figure 5.1: Labels in the bit-reversal ring with $n = 32$

$000, 100, 010, 110, 001, 101, 011, 111$ (in binary) when $n = 8$. Figure 5.1 gives a picture of what this looks like for $n = 32$.

For n not a power of 2 we look up Frederickson and Lynch [FL87] or Attiya *et al.* [ASW88]. In either case we get $\Omega(n/k)$ order-equivalent members of each equivalence class after k active rounds, giving $\Omega(n/k)$ messages per active round, which sums to $\Omega(n \log n)$.

For non-comparison-based algorithms we can still prove $\Omega(n \log n)$ messages for time-bounded algorithms, but it requires techniques from **Ramsey theory**, the branch of combinatorics that studies when large enough structures inevitably contain substructures with certain properties.[5] Here "time-bounded" means that the running time can't depend on the size of the ID space. See [AW04, §3.4.2] or [Lyn96, §3.7] for the textbook version, or [FL87, §7] for the original result.

The intuition is that for any fixed protocol, if the ID space is large enough, then there exists a subset of the ID space where the protocol

[5]The classic example is **Ramsey's Theorem**, which says that if you color the edges of a complete graph red or blue, while trying to avoid having any subsets of k vertices with all edges between them the same color, you will no longer be able to once the graph is large enough (for any fixed k). See [GRS90] for much more on the subject of Ramsey theory.

acts like a comparison-based protocol. So the existence of an $O(f(n))$-message time-bounded protocol implies the existence of an $O(f(n))$-message comparison-based protocol, and from the previous lower bound we know $f(n)$ is $\Omega(n \log n)$. Note that time-boundedness is necessary: we can't prove the lower bound for non-time-bounded algorithms because of the $i \cdot n$ trick.