Leader Election in Synchronous Graphs with Constant-Size Messages

based on

Deterministic Leader Election in $\mathcal{O}(D + \log n)$ Time with Messages of Size $\mathcal{O}(1)$

Jakub Oskwarek

January 2024

Abstract

This project aims to understand the leader election algorithm proposed by Casteigts, Métivier, Robson, and Zemmari, to outline the necessary proofs for it, and to implement it in Go.

1 Network properties

The algorithm [1] is applicable to any connected graph of n nodes, m edges, and diameter D, as long as the system adheres to the following assumptions:

- Every node has a **unique identifier**, which is a positive integer. For simplicity of analysis, we also assume that the size (i.e. number of digits) of each such identifier is logarithmic with respect to n, but the algorithm itself does not exercise this property. **The node with the greatest identifier is supposed to become the leader and the entire network should know that identifier by then.**
- The algorithm proceeds in so-called *bit rounds*, i.e. synchronous rounds in which the nodes exchange messages consisting of single bits. In practice, without loss of generality, the messages can be thought of as simply having **constant size**, i.e. independent of n, m, and D.
- The communication channels are **fully reliable** — messages are neither lost nor mangled in transmission.

The constant-size requirement is particularly limiting — it means that no identifier can be directly transmitted. It also means that the algorithm is suitable for systems where time is not a critical resource but data transfer is. Of course, the goal is to beat the naïve algorithm in which every node simply broadcasts its identifier bit by bit in $\mathcal{O}(D \log n)$ rounds. In fact, the authors on the original paper recall other scientific results to conclude that their $\mathcal{O}(D + \log n)$ solution is asymptotically optimal in this sense.

2 The general idea

Most terms and variable names have been kept the same as in the original paper.

The naïve approach of broadcasting identifiers bit by bit is not as misguided as it might appear. After all, the leader's identifier has to eventually reach every node somehow. However, we can improve this method by gradually making the other nodes realize that they stand no chance of becoming the leader and forcing them to abandon their own broadcast in favor of advocating for more prominent candidates. In particular, every node will operate on bit strings to build up the best *prefix* it has seen around, and keep its neighbors up to date on that.

In order to be able to quickly rule out certain prefixes, the idea of *α-encoding* is introduced:

$$\alpha(q) = 1^{|q_{(2)}|} 0 q_{(2)}$$

where q is an identifier. In other words, the α-encoding of an identifier consists of as many ones as there are digits in the binary representation of this identifier, then a zero, and then the binary representation itself. For example, 5 is 101 in binary, and 101 has three digits, so there will be three ones, then a zero, and then 101 itself:

$$\alpha(5) = 1110101$$

This encoding is very useful because instead of transmitting the integers bit by bit and comparing them as numbers, which is quite cumbersome, we can transmit their α-encodings character by character and compare them as strings, i.e. simply detect a difference at the first mismatched symbol. Indeed:

- if two identifiers differ in length of their binary representation, the shorter (smaller) one will have a 0 at a position where the longer (bigger) one still has a 1 (from the opening chain of 1's), so the bigger one will correctly compare as greater.
- if the binary representations of two identifiers have the same length, the opening chains of 1's and the single 0's will match and then the binary representations will themselves be correctly compared lexicographically.

Thus, every node will maintain a variable called `Prefix` that represents the best prefix it has seen (initially empty), and a boolean flag `Active` that will start as `true` but become `false` when a better prefix than this node's own is encountered (this corresponds to the realization that it cannot win the election). A copy of every neighbor's `Prefix` will also be kept. In every round, the node will look at those copies, the round number, and its own α-encoded identifier, and, based on that, it will modify its `Prefix` using one of seven available operations. It will then announce that operation to its neighbors (using a constant-size message) so that they can modify their copy in the same way. Naturally, it will then process the incoming messages and modify local copies. The authors call this part *the spreading algorithm* $\mathcal{S}$.

During the construction of the prefixes, a spanning tree of the graph will also emerge: every node will declare itself to be a *child* of the neighbor that forwards the currently best prefix. For that purpose, every node will maintain a variable called `Parent` and remember its own children. Eventually, the only node with `Parent` not set will be the leader, who will naturally become the root. With this extension, the algorithm is called $\mathcal{ST}$.

Finally, every node will maintain a boolean variable called `Termination`. When a node sets it to `true`, it means that no modifications concerning the prefixes or parent/child dependencies are occurring in the subtree of that node. However, the node must not shutdown yet, as it is not permanent — it can change back to `false` when a better prefix arrives or when a new child appears. The `Termination` variable is also reported to the neighbors. Only when a stable report of `Termination` = `true` reaches an `Active` node from *all* its neighbors, this node declares itself the leader and broadcasts the information that the election has concluded — `shutdown`. When a node receives such message, it becomes `Done`, and forwards `shutdown` in the next round. This complete algorithm is called $\mathcal{STT}$.

To summarize, in general, a single round on a given node u will look like the following:

Algorithm 1 A single round

```
procedure PROCESS($r$)                                  ▷ $r$ is the round number
    operation ← UPDATEPREFIXANDPARENT($t$)
    ▷ Announce own modifications to neighbors ◁
    for $v \in N(u)$ do
        if $v$ is Parent then
            SEND (operation, true) TO $v$               ▷ true means "You are my parent"
        else
            SEND (operation, false) TO $v$
    if operation = shutdown then
        EXIT                                            ▷ The processing of this node is finished
    PROCESSINCOMINGMODIFICATIONS()
    ▷ Announce own Termination ◁
    for $v \in N(u)$ do
        SEND (Termination) TO $v$
    PROCESSINCOMINGTERMINATIONS()
    UPDATETERMINATION()
    ▷ Report own Termination to Parent ◁
    if Parent is set then
        SEND (Termination) TO Parent
    COLLECTTERMINATIONSFROMCHILDREN()
```

The procedures PROCESSINCOMINGMODIFICATIONS and PROCESSINCOMINGTERMINATIONS will not be presented in full detail because they are not very interesting — they simply take note of the messages in order to maintain up-to-date local copies of the neighbor's states, keep track of parent/child dependencies, and occasionally set `Termination` ← `false` (when a new child appears) or `Done` ← `true` (when the received `operation` is `shutdown`).

Note that the communication happens in three phases — one for `operation`s and two for `Termination`s. I implemented it this way in order to clearly separate

- prefix & tree updates
- termination updates that result from gaining children

- termination updates that result from internal re-computation

I believe, however, that this could be accumulated in order to reduce the number of messages by a constant factor. Moreover, many of these messages could be `nil`, indicating that nothing has been modified, but since the messages are constant-size anyway, I decided that, once again the total symmetry would be clearer.

3 The `Prefix` and `Parent` updates

The authors of the original paper propose five rules for deciding how a node u should update its prefix. The implementation of UpdatePrefixAndParent consists almost solely of checking those rules top to bottom and taking the first one that matches. If all the rules fail, the resulting operation is `null` and no modification is carried out.

The notation P_u is used for the value of u's `Prefix` variable.

Rule 1. Delete

This rule has two versions and both of them check all the neighbors $v \in N(u)$. If any characters at all can be deleted, we pick the version and neighbor that deletes the most.

The resulting operation for u is one of `delete1`, `delete2`, and `delete3`.

(a) If u has a neighbor v who has recently deleted and P_v is a proper prefix of P_u, then u deletes as many "extra" characters from the end of P_u as possible, but no more than 3.

Intuitively: our neighbor v has recently corrected its suffix and we are similar, so we try to do the same, within sensible limits.

Example: if $P_u = 110$, $P_v = 11$, and v deleted something in the previous round, then we set $P_u \leftarrow 11$ and the `operation` is `delete1`.

(b) If u has a neighbor v such that $P_u = c0s$ and $P_v = c1\ldots$ (with c being an unspecified bit string), then u simply deletes the extra suffix s, if it is nonempty.

Intuitively: our neighbor v knows a prefix that has a 1 at a position where we have 0, so the prefix we know cannot be the leader's prefix. Therefore, we certainly have to get rid of the part after the 0, because it must be wrong.

Example: if $P_u = 11011$ and $P_v = 111\ldots$, then we set $P_u \leftarrow 110$ and the `operation` is `delete2`.

It can be shown that such deletion is safe, i.e. s will never have more than 3 characters.

Rule 2. Change

If u has a neighbor v such that $P_u = c0$ and $P_v = c1\ldots$, then u sets

$$\begin{aligned} P_u &\leftarrow c1 \\ \texttt{Parent} &\leftarrow v \\ \texttt{Active} &\leftarrow \texttt{false} \end{aligned}$$

Intuitively: our neighbor knows a slightly better prefix, so we should change ours to match it better. We also direct ourselves towards that better prefix and realize that we are not the leader.

Example: if $P_u = 110$ and $P_v = 111\ldots$, then we set $P_u \leftarrow 111$, update u's `Parent` and `Active`, and the `operation` is `change`.

Notice how Rule 1. seems to be a preparation for Rule 2. This will, in fact, be formalized.

Rule 3. Append 1

If u has a neighbor v such that P_v starts with P_u and then a 1 follows, then u sets

$$\begin{aligned} P_u &\leftarrow P_u 1 \\ \texttt{Parent} &\leftarrow v \end{aligned}$$

Intuitively: our neighbor knows a slightly longer (and therefore, better) prefix, so we decide to copy it and direct ourselves towards this smart neighbor.

Example: if $P_u = 11$ and $P_v = 111$, then we set $P_u \leftarrow 111$, update u's parent, and the `operation` is `append1`.

Rule 4. Append 0

Analogous to Rule 3.

Rule 5. Extend

The resulting operation for u is one of `append1` and `append0`.

If `Active`, with the round number being r, u appends the r-th character from its α-encoded identifier to P_u.

Intuitively: we still live in the blissful belief that we stand a chance of becoming the leader, so we treat our own identifier as the best one so far.

Example: if u is `Active` in round $t = 2$, having the identifier 3 α-encoded to 11011, and $P_u = 1$, then we set $P_u \leftarrow P_u 1 = 11$, and the `operation` is `append1`.

4 The `Termination` updates

As mentioned earlier, intuitively, `Termination` = `true` at node u means that there is nothing interesting going on in the subtree of this node, i.e. the prefixes and the structure of this tree have (temporarily) stabilized, and, if nothing new happens, it is ready to finish the execution of the algorithm. Naturally, whenever one of the five rules matches at u or u gains a new child, we set `Termination` $\leftarrow$ `false`. The only place where `Termination` is set to `true` for u is the procedure UpdateTermination(), and only if the following conditions are met:

- `Active` is set to `false` (a node that does not even know whether it has a chance to win cannot deem its situation stable) — notice that the actual leader will never have `Termination` = `true`
- `Prefix` represents a valid α-encoded identifier, i.e. has the form $\underbrace{1 \ldots 1}_{k \text{ ones}} 0 \underbrace{0 \ldots 1}_{k \text{ digits}}$ (otherwise, the tree cannot possibly be ready, because it certainly does not know the leader)
- u and all its neighbors have the same `Prefix` (otherwise, someone must be mistaken about the leader's identity)
- all of u's children have `Termination` = `true` (a natural recursive requirement)

In CollectTerminationsFromChildren, the node checks whether all its neighbors have announced `Termination` = `true`, and receives the reports from children — if their `Termination` is still `true`, the node knows, that it is the leader.

5 Proof outline

This section presents the highlights of the proof from the original paper [1]. I do not always employ the highest degree of formalism, but I believe that the necessary intuition is clear.

Lemma 1. *If a `delete` operation happens in one round at u, there will be a `delete` or a `change` in the next round.*

Proof. Inductive.

1. Base: $r = 1$. Rule 1. is the only one that produces `delete` operations, and it requires either of:

   (a) a `delete` in the previous round (but there was no previous round)

   (b) non-empty `Prefix`es (but all are initially empty)

   Therefore, a `delete` is impossible if $r = 1$, so the implication holds.

2. Step. We have to branch depending on which version of the rule was used.

   (a) The neighbor v that allowed this rule to match in round r, must have deleted in round $r - 1$. The inductive hypothesis tells us that it deleted or changed in round r.

   We know that at the start of round r, P_v was a proper prefix of P_u, so if v deleted again, then the shortened P_v is still a proper prefix of the shortened P_u, and v still allows Rule 1a. to match in round $r + 1$, so u will also delete again.

   If, on the other hand, v decided to change, then, during round r, P_v changed from $p0$, to $p1$ (where p simply denotes an unspecified prefix). Again, we know that P_v used to be a proper prefix of P_u, so, during round r, P_u was truncated from something of the form $p0s$ to something of the form $p0t$, where t is a (possibly empty) proper prefix of s. If t is not empty, then $P_v = p0$ is still a proper prefix of $P_u = p0t$ and Rule 1a. will match again in round $r + 1$, making u use `delete`. However, if t happens to be empty, we have a guaranteed match of Rule 2. because simply $P_u = s0$, which, combined with $P_v = s1$, results in a `change` at u.

Therefore, if Rule 1a. was exercised in round r, the lemma indeed holds.

(b) Here, P_u started round r as $c0s$ and dropped the s to become $c0$, whereas P_v started as $c1\ldots$ and performed an unspecified operation. However, only two scenarios are possible:

- v used `delete` and lost the 1 and everything after it — then, P_v certainly became a proper prefix of $P_u = c0$ and Rule 1a. matches, implying a `delete` at u in round $r + 1$.
- v somehow kept the 1, which guarantees that Rule 2. will match in round $r + 1$ and u will use `change`.

Thus, the lemma also holds if Rule 1b. matched in round r.

□

From lemma 1. we draw an important observation: that every node, except for the leader, will behave as follows:

(i) use Rule 5. a few times to extend `Prefix` with the initial bits of the α-encoded identifier

(ii) use `delete` a few (maybe zero) times

(iii) use `change` to create a `Prefix` which is better than the one before the `delete`s

We also observe that once a node's `Prefix` becomes the leader's α-encoded identifier, it is never modified again because modifications would have to yield an even better `Prefix`, but one does not exist. In particular, the leader will simply transfer its α-encoded identifier to its `Prefix` and then it will stop modifying.

Lemma 2. *If I is the α-encoding of the greatest identifier, then all nodes whose distance to the neighbor is d or smaller, will know I after $|I| + 6d$ rounds.*

Proof. Inductive.

1. Base: $d = 0$. The only node with such distance is the leader node itself, and the only thing it needs to do is move I to `Prefix` bit by bit. This happens in $|I| = |I| + 6 \cdot 0$ rounds and the lemma holds.
2. Step. Since the network is connected, a node u whose distance from the leader is $d+1$ has a neighbor v whose distance is d. The inductive hypothesis tells us that this neighbor learns I in $|I| + 6d$ rounds. Here, the authors of the original paper employ a separate induction and an exhaustive case analysis (quite long and not very interesting, therefore omitted) in order to prove that the updates of `Prefix`es work in such a way that neighbors always differ by no more than six characters at the end. Therefore, since v's `Prefix` is already I, it will not change, and u can slowly take six rounds to match it. This gives $|I| + 6d + 6 = |I| + 6(d + 1)$ rounds and the lemma holds.

□

Lemma 3. *When the entire network knows the leader, the `Parent` variables determine a valid spanning tree.*

Proof. Let us track the idea of a node's "personal best" at round r, i.e. the snapshot of best `Prefix` it has had up to this round (inclusive) and the number of the first round in which this prefix appeared. "Personal bests" have a natural ordering: one "personal best" is better than the other if it recorded a better `Prefix` or an equally good `Prefix` that appeared earlier. Also note that only `delete` operations cause the current `Prefix` to become worse, so by *the best `Prefix`* we really mean *the `Prefix` obtained by the last non-`delete`.*

By construction, we see that the only places where `Parent` changes are Rules 2., 3., and 4., and in every one of those places, during round r, a node u chooses a `Parent` that has a better `Prefix` than u itself could ever have before. By doing that, u has just set a new "personal best", which is, however, necessary worse that the parent's "personal best" because even if the `Prefix`es are now equally good, the parent obtained such a `Prefix` (or even a better one) earlier. This means that, following the `Parent` links, we encounter increasingly better "personal bests". Thus, the structure of those links must be acyclic.

Furthermore, nodes with `Active` = `false` must have parents. Since we assumed everyone knows the leader, the only remaining node with `Active` = `true` is the leader itself, and, intuitively, all the other nodes can reach it through their `Parent` links.

Therefore, the resulting structure is indeed a spanning tree. □

Lemma 4. *The leader node (and only this node) will eventually broadcast `shutdown`.*

Proof. Someone will certainly broadcast `shutdown`, since after everyone knows the leader, no modifications happen and waves of `Termination` = `true` start to flow. To ensure that this will not happen to early, we have to show that only the leader will be able to actually order a `shutdown`.

This will be a proof by contradiction. Let us assume that some other node u ordered a `shutdown`. For that, all the neighbors had to have the same `Prefix` as u and to announce `Termination` = `true`. We consider all the nodes that ever had u's α-encoded identifier as their `Prefix`. There must have once been a tree rooted at u, spanning all those nodes. Intuitively, because of the recursive condition for terminations, the requirement of having u's identifier as `Prefix` and `Termination` = `true` "floods" the entire network along the edges of that tree. The actual leader is included in this madness! But it cannot possibly have set `Termination` = `true`, seeing u's identifier, because it knows a better one — its own! □

All these lemmas and observations give us the following theorem:

Theorem 1. *The algorithm's running time is $\mathcal{O}(D + \log n)$ (bit) rounds.*

Proof. The identifiers have length $\mathcal{O}(\log n)$ and so do their α-encodings. From lemma 2. we know that the entire network will learn the leader's α-encoded identifier in $\mathcal{O}(D + \log n)$ because all nodes are at distance D or closer to the leader. The wave of `Termination` = `true` will reach the leader from the bottom of the spanning tree described in lemma 3., taking $\mathcal{O}(D)$ rounds. Finally, the `shutdown` order from lemma 4. will travel for $\mathcal{O}(D)$ rounds. This gives the total number of rounds:

$$\mathcal{O}(D + \log n)$$

The number of messages is $\mathcal{O}((D + \log n)m)$ □

References

[1] Arnaud Casteigts, Yves Métivier, John Michael Robson, and Akka Zemmari. Deterministic leader election in $\mathcal{O}(D+\log n)$ time with messages of size $\mathcal{O}(1)$. *arXiv: Distributed, Parallel, and Cluster Computing*, 2016.