Humblet's clique leader election algorithm

Our goal is to elect a leader in a complete graph with n nodes and asynchronous first-in-first-out communication. We assume that every node has a unique identifier, and that all identifiers are totally ordered. Initially, no node needs any knowledge about the identities of the nodes at the other ends of its incident edges, and these edges may be arbitrarily ordered.

The simplest solution to this problem is to let every node send its identifier to all its neighbors, and then let every node select the neighbor with the largest identifier. This approach requires $O(n^2)$ messages. Here we describe an algorithm developed by Pierre Humblet in 1984, which solves this problem and only requires $O(n \lg n)$ messages.

Algorithm description

The idea is that every node tries to "capture" its consecutive neighbors and expand its "territory", until it is either defeated by another node or its territory is large enough so that it can announce itself as a leader.

Every node is in one of three states: *active*, *inactive* or *captured*. Every node remembers its *level*, i.e., the number of nodes it has already captured. Initially, all nodes are active. An active node tries to capture its consecutive neighbors by sending a `capture` message, waiting for an `accept` response, increasing its level, and repeating this for the next node. A capture may fail, and in this case no `accept` response is sent and the node waiting for it may not proceed. The `capture` message contains the sender's identifier and current level.

When an active or inactive node receives a `capture` message, it compares the sender's pair (*level*, *identifier*) with its own (they must be different, because the identifiers are unique), and if the sender's pair is greater, it responds with an `accept` message and becomes captured. Otherwise, no response is sent and the sender may not proceed. A node may be captured multiple times and it remembers its *owner*, i.e., the last node which captured it.

When an already captured node receives a `capture` message, it forwards this message to its owner (which is not necessarily still active) and waits for a response with the result of comparison between the sender and the owner. During this time, any other incoming `capture` messages are queued and only processed after the awaited response, which may potentially result in a change of the owner. If the sender has won the comparison, the captured node changes its owner to the sender and responds to it with an `accept` message. In this case, if the previous owner was active, then it becomes inactive, which means that it no longer tries to capture next nodes, but is not captured yet. Otherwise, if the sender has lost the comparison, the captured node keeps its owner and sends no response to the sender, which then may not proceed. The queueing of `capture` messages prevents a situation in which a node is about to change its owner, but the `yes` message has not yet arrived, and the node erroneously forwards a message to its previous owner.

When an active node reaches a level such that its *territory*, i.e., the node itself and all its captured nodes, is a majority of all nodes (which means that $level + 1 > n/2$), then this node announces itself as a leader and finishes. All other nodes then immediately finish after receiving this announcement.

Implementation details

Every node keeps the following variables:

- `id` – the node's identifier
- `size` – the size of the graph
- `active` – whether the node is active; initially `true`
- `level` – the node's level; initially 0
- `owner` – the index (neighbor number) of the node's owner; if `none`, then this node is not yet captured
- `queue` – the queue for `capture` messages
- `contender` – the index of the node which sent a `capture` message when this node was already captured; only set for the time of waiting for a response from the owner; the `queue` may only be processed when this variable is `none`

A node is active if and only if `active && owner == none`, inactive if and only if `!active && owner == none`, and captured if and only if `!active && owner != none`.

Every node starts by sending a `capture` message to its first neighbor. Then it enters a loop in which it processes the incoming messages.

After receiving a `capture` message, the node checks if it is forwarded from a captured node and enqueues this message. The message is forwarded if and only if the index of the sender is less than the node's current level, because a captured node only forwards a message to its owner, and a non-forwarded message must come from a non-captured node.

After receiving an `accept` message, the node increases its level, and if it is active, checks whether to capture next node or announce itself as a leader and finish.

When receiving a `yes` or `no` message, the node must have been already captured, must have received a `capture` message from `contender`, and this must be a response from the owner. If the response is `yes`, the node changes its owner to `contender`, sends `accept` message to it, and sets the `contender` variable to `none`. If the response is `no`, it ignores the `contender` and only sets this variable to `none`. After this, the processing of the `queue` may continue.

After receiving a `leader` message, the node saves the leader's identifier and finishes.

After processing the incoming message, and while the `contender` variable is `none`, the node processes the `queue`.

If the dequeued `capture` message is forwarded, the node responds with `yes` or `no` depending on the result of the comparison with the original sender. If the sender has won, the node sets `active` to `false`, which either makes it inactive or keeps it captured.

If the dequeued `capture` message is not forwarded and the node is not captured, then the node compares itself with the sender. If the sender has won, then the node becomes captured and responds with `accept` message to the sender. Otherwise, the node ignores the sender.

If the dequeued `capture` message is not forwarded and the node is captured, then the node forwards the message to its owner and sets the `contender` variable to the sender. This variable can then only be unset after receiving a `yes` or `no` response from the owner, and during this time the `queue` may not be processed.

Correctness proof

A node is a *winner* when it has won its last capture attempt and a relevant `yes` or `accept` message is currently in transit.

A node's *potential level* is its level when it is not a winner, or its level plus one when it is a winner.

Lemma 1. *After a node loses its capture attempt, neither its level nor its potential level can increase.*

Proof. The only way for the node's level to increase is by receiving an `accept` message, which can only happen after winning a capture attempt. The only way for the node's potential level to increase is by increasing its level or by winning a capture attempt. □

Lemma 2. *After a node with level l stops being active, its potential level can only become at most $l + 1$.*

Proof. The only way for the node's level to increase is by receiving an `accept` message, which in turn requires a `capture` message to be sent. When the node becomes a winner, its potential level becomes $l + 1$, and it will eventually receive an `accept` message, which will make both its level and potential level equal $l + 1$. It will not however send a next `capture` message, so neither its level nor its potential level will increase. □

A node is a *candidate*, when it is active and has not lost its last capture attempt, so either it is a winner or the `capture` message is still in transit.

Lemma 3. *Once a node stops being a candidate, it can not become a candidate again.*

Proof. A node can not become active again, and an ignored `capture` message prevents the node from sending any other `capture` messages. □

Lemma 4. *At any moment at least one node is a candidate.*

Proof. Since all identifiers are unique, at any moment there must be a node x with the largest (*potential level*, *identifier*) pair. Suppose that this node is not a candidate. Then either it has stopped being active or it has lost its last capture attempt. If it has lost its last capture attempt, then at that moment some node y must have had larger (*level*, *identifier*) pair. By lemma 1, the x's (*potential level*, *identifier*) pair could not have increased afterwards, so y's pair must still be larger, but x's pair was assumed to be currently the largest (contradiction). If it has stopped being active, then it must have been defeated by some node y. Let i_x and i_y be the identifiers of x and y respectively. Let l_x and l_y be the levels of x and y respectively, at the moment when y defeated x. Let p_x and p_y be the current potential levels of x and y respectively. Since y has won the capture attempt, there must be $(l_y, i_y) > (l_x, i_x)$ and $p_y \geq l_y + 1$. By lemma 2, there must be $p_x \leq l_x + 1$. Thus, $(p_y, i_y) \geq (l_y + 1, i_y) > (l_x + 1, i_x) \geq (p_x, i_x)$, but x's pair was assumed to be currently the largest (contradiction). Therefore, the node x must be a candidate. □

Lemma 5. *Some node will announce itself as a leader.*

Proof. By lemma 3 and lemma 4, some node will be a candidate for the whole run time of the algorithm. This node will eventually reach a level high enough to announce itself as a leader. □

Lemma 6. *If a node x_1 reaches level l at moment t_1, and another node x_2 reaches the same level l at moment t_2, then x_1's territory from moment t_1 is disjoint with x_2's territory from moment t_2.*

Proof. Assume, without loss of generality, that t_1 is before t_2. Suppose that some node belongs both to x_1's territory at moment t_1 and to x_2's territory at moment t_2. Then it must have been captured by x_2 in the time between t_1 and t_2. At that moment x_1 has already reached level l, but x_2 has not yet reached this level. Therefore, x_2 must have lost this capture attempt (contradiction). □

Lemma 7. *At most $n/(l+1)$ nodes can reach level l.*

Proof. Consider the set of all nodes which reached level l. Every one of them had territory of size $l+1$ at the moment of reaching this level. By lemma 6, all these territories must be mutually disjoint, so this set can have at most $n/(l+1)$ elements. □

Lemma 8. *At most one node can announce itself as a leader.*

Proof. A node can only announce itself as a leader when its level l is greater than $n/2-1$. Then, $n/(l+1) < n/(n/2) = 2$, so by lemma 7, at most one node can reach this level. □

Theorem 1. *Exactly one node will announce itself as a leader.*

Proof. Follows from lemma 5 and lemma 8. □

Complexity analysis

Lemma 9. *Every* `capture` *message sent from an active node generates at most three other messages.*

Proof. A `capture` message may be forwarded to the receiver's owner, the owner does not forward this message any further and responds with `yes` or `no` message, and then an `accept` response may be sent. □

Theorem 2. *At most $4nH_{\lfloor n/2 \rfloor}+n-1$ messages are sent, where H_i is the i-th harmonic number.*

Proof. Group all `capture` messages from active nodes by the sender's level. An active node may only sent one `capture` message per level. The maximum level at which an active node may send a `capture` message is $\lfloor n/2 \rfloor - 1$. By lemma 7 and lemma 9, the total number of messages generated by capture attempts is at most

$$\sum_{l=0}^{\lfloor n/2 \rfloor - 1} 4n/(l+1) = 4n \sum_{l=1}^{\lfloor n/2 \rfloor} 1/l = 4nH_{\lfloor n/2 \rfloor},$$

and the final leader announcement generates exactly $n-1$ messages. □

References

P. Humblet, "Selecting a leader in a clique in $O(N \log N)$ messages," The 23rd IEEE Conference on Decision and Control. IEEE, Dec. 1984. doi: 10.1109/cdc.1984.272191.

N. Santoro, "Design and Analysis of Distributed Algorithms," Wiley Series on Parallel and Distributed Computing. John Wiley & Sons, Inc., Nov. 22, 2006. doi: 10.1002/0470072644.