Alistarh et al. (2015) — "How to Elect a Leader Faster than a Tournament"

November 21, 2017; rev. November 23, 2017
Lily Li

1 Introduction

Problem Definition. An algorithm which solves leader election among n processes must satisfy: (1) termination - every process p_i must eventually output win or $lose$ and (2) unique winner - there is exactly one process which outputs win. Further, no process may lose before the eventual winner starts its execution and the algorithm must be linearizable; the first operation is win and all subsequent operations is $lose$. In-class, we have a seen an leader election algorithm which builds a *tournament tree* with n leaves. The step and message complexities of this decades old algorithm are $\Theta(\log n)$ and $\Theta(n^2 \log n)$ respectively.

Model. We have a complete asynchronous message passing system with at most $f < \lceil n/2 \rceil$ faulty processes. Further we assume a strong adversary which can examine the systems state and the outcome of random coin flips adjusting the schedules of steps, message delivery events, and failures.

Results. In this work by Alistarh et al. presented at PODC 2015, the authors present a leader election algorithm which requires $O(\log^* n)$ steps by each process and $O(n^2)$ point-to-point messages. They pair this with a lower bound of $\Omega(n^2)$ message, proving optimal message complexity.

2 Leader Election Algorithm

In the code we will make use of the high-level protocol COMMUNICATE. Let p be a procedure which consists of broadcasting a message and waiting for a response. COMMUNICATE$\langle p \rangle$ will wait for at least $\lceil n/2 \rceil + 1$ responses before proceeding (this is reminiscent to what we did when we tried to simulate single-reader single-writer registers in a message passing system). We call a set of $\lceil n/2 \rceil + 1$ processes a **quorum** since these arise often in our analysis. A process can execute the following operations:

broadcast(v): process p_i broadcasts value v to all other processes (including itself). If process p_j receives broadcast$v$ from p_i, it updates $S_j[i] \leftarrow v$.

collect$()$: process p_i sends a collect message to all other processes (including itself). If process p_j receives collect$()$ from p_i, it sends message $\langle S_j \rangle$ to p_i.

random(p): (local operation) process p_i receives 1 with probability p and 0 otherwise.

What we want is an algorithm which iteratively reduces the number of participating processes. At each round processes flip a random coin and output $lose$ or $survive$ depending on the result. Surviving processes will try again in the next round. A naïve implementation would fail since the adversary can schedule the "losing" processes first.

2.1 Poison Pill (Homogeneous)

See Algorithm 1. Process p_i has two local variables: a vector S_i of length n which record the states of processes (with all entries initially set to w) and an $n \times n$ matrix V_i storing the view as seen by other processes (initially empty). States can be any of w (waiting), c (commited), 0 (lose), and 1 (survive). A process p_j is **active** from the perspective of p_i if column j in V_i contains at least one entry of status 1 or c and no entries of status 0.

Claim 1. *If all processes return then some process outputs* 1.

Proof. Suppose for a contradiction that all processes output 0. If a process receives 1 upon executing random $\left(\frac{1}{\sqrt{n}}\right)$, then it must output 1 so all processes received 0. Let process p_i be the last to complete the broadcast operation on Line 5. When p_i's broadcast operation finishes, the 0 of every process gets stored by a quorum. Consider V_i at the completion of Line 6. Every column must have one 0-entry since the

Algorithm 1 Homogeneous Poison Pill: code for process p_i.

Initialize $S_i[1...n] = [w, ..., w]$ and $V_i[1...n][1...n] = \emptyset$
$S_i[i] \leftarrow c$
$\mathsf{COMMUNICATE}\langle\mathsf{broadcast}(S[i])\rangle$
$S_i[i] \leftarrow \mathsf{random}\left(\frac{1}{\sqrt{n}}\right)$
$\mathsf{COMMUNICATE}\langle\mathsf{broadcast}(S_i[i])\rangle$
$V_i \leftarrow \mathsf{COMMUNICATE}\langle\mathsf{collect}()\rangle$
if $S_i[i] = 0$ and $\exists k : p_k$ is active **then**
    return 0
end if
return 1

set of processes that stored 0 must overlap with the set of processes which successfully sent their status vector to p_i. Thus p_i must output 1. ☐

Claim 2. *The expected number of processes that return* 1 *is* $O(\sqrt{n})$.

Proof. Suppose p_i received 1 from its execution of random at t_i. Then all process which received 0 at any time $\geq t_i$ must output 0. Consider such a process p_j. Observe that at t_i, $S_i[i] = \top$ has been broadcast to a quorum. Thus the set of processes which received $S_i[i]$ and the set of processes which sends their status to p_j upon its execution of collect() overlaps. p_j will see that p_i has $\top$ or 1 and will return 0. Now simply observe that the expected number of 1s as well as the expected index for the first 1 are both $\sqrt{n}$. ☐

2.2 Poison Pill (Heterogeneous)

It can be shown that there will be $\Omega(\sqrt{n})$ survivors at every round of the homogeneous algorithm — the probability of getting 1 is fixed. To improve this bound we need to change this probability.

Algorithm 2 Heterogeneous Poison Pill: code for process p_i.

$S_i[i] \leftarrow \{c, \{\}\}$
$\mathsf{COMMUNICATE}\langle broadcast(S_i[i])\rangle$
$V_i \leftarrow \mathsf{COMMUNICATE}\langle\mathsf{collect}()\rangle$
$t \leftarrow \{j : \exists k : V_i[k][j] \neq \perp\}$
$p \leftarrow \frac{\log|t|}{|t|}$ # $p \leftarrow 1$ if $|t| = 1$
$S_i[i] \leftarrow \{\mathsf{random}(p), t\}$
$\mathsf{COMMUNICATE}\langle\mathsf{broadcast}(S_i[i])\rangle$
$V_i \leftarrow \mathsf{COMMUNICATE}\langle\mathsf{collect}()\rangle$
if $S_i[i][0] = 1$ and $\exists k : p_k$ is active **then**
    return 0
else
    return 1
end if
return 1

2.3 Analysis

Now we just need to show that the expected number of processes which survive using the heterogeneous poison pill algorithm is $\Theta(\log n)$.

Claim 3. *Let S be the set of processes which flipped* 0 *and survived and let U be the union of all the elements that they found not waiting. Then for $i \in U$, every $j \in t_i$ is in U.*

Proof. Intuitively this should make sense as U is pretty comprehensive (we are taking the union of a bunch of stuff). Since a process p_i which flipped zero can only survive if is sees only other processes which lost, every $p \in U$ obtained 0. There is some entry of the view V_i corresponding to p which has 0. Further it also has the list associated with p. This list is taken into account during the union thus p_i see all the elements in the list t of p. ☐

Claim 4. *If processor q completed the first broadcast call no later than p completed its first propagate call, then q will be included in the l list of p.*

Claim 5. *The maximum expected number of processors that flip 0 and survive is $O(\log n) + O(1)$.*

Proof. Let S be the set of z processes. And U be defined as above. By Claim 4 and Claim 3, it must be the case that if $p \in U$ and q finished the first broadcast no later than p, then $q \in U$. Notice that if we order the processes by the time they finish the first broadcast, all processes in U must come before the processes not in U.
Since U forms a closed set and all processes in U flipped 0, the probability that each process flipped 0 is at most $\left(1 - \frac{\log |U|}{|U|}\right)$. The probability for all processes in U to flip 0 is $\left(1 - \frac{\log |U|}{|U|}\right)^{|U|} = O\left(\frac{1}{|U|}\right)$. Since $z \in U$, This is $O\left(\frac{1}{z}\right)$. □

Claim 6. *The maximum expected number of processors that flip 0 and survive is $O(\log^2 n) + O(1)$.*

Proof. Order the processes according to the time they finish their first broadcast. By property 4, the process ordered first has $|t| \geq 1$, the process ordered second has $|t| \geq 2$ and so on. Since the probability of flipping 1 decreases as $|t|$ increases, the best expectation achievable is

$$1 + \sum_{l=2}^{n} \frac{log|t|}{|t|} \in O(\log^2 n).$$

□

Reference

Alistarh, D., Gelashvili, R., and Vladu, A. (2015). How to elect a leader faster than a tournament. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 365–374. ACM.