Election Algorithms

Chapter 7
1997-12-12
What is an election?

“to start from a configuration where all processes are in the same state and arrive at a configuration where exactly one process is the leader”

Goal: to obtain low message complexity
Elections are needed...

• when a *centralized* algorithm is to be executed in a distributed system
• for instance, after a system crash when it is unknown which nodes are working
Definition of Election Alg.

1. Each process has the same local algorithm
2. The algorithm is *decentralized*, i.e., a computation can be initiated by an arbitrary non-empty subset of nodes
3. The algorithm reaches a terminal config. in each computation, and in each such config. there is exactly one process in the state *leader* and all others are in the state *lost*
Possible states of the nodes

- sleep - before it has executed
- candidate - doesn’t know if it was elected
- leader - knows it won the election
- lost - will not become leader
Assumptions about the distributed system

1. Fully asynchronous - no common clock, arbitrary transmission time

2. Each process has a unique identity $p \in P$ known to the process. $P$ is totally ordered by relation $\leq$ on $P \times P$

3. Each message may contain $O(w) = O(\log |P|)$ bits.
Elections and Waves

Elections elect the process with the lowest pid

- Th. 6.11: Every INF algorithm is a wave alg.
- Th. 6.12: Every wave algorithm can be used to compute INF

TREE, PHASE, FINN are decentralized
The TREE algorithm

- Requires at least all leaves are initiators, therefore first a *wakeup* phase
- send wakeup only once to each process
- **While** # of *wakeup* received < #neighbours **then** receive *wakeup*
  - **Do** receive suggestions **until** only one neighbour hasn’t sent suggestion
  - **When** only q₀ is left **then** send q₀ the best suggestion
  - **Wait** for last suggestion from q₀
  - **If** my pid = the smallest suggestion **then** I’m *leader*, **else** I’m *lost*
  - send smallest suggestion to all neighbours ≠ q₀
\begin{verbatim}
var wsp : boolean init false;
wr_p : integer init 0;
rec_p[q] : boolean for each \( q \in \text{Neigh}_p \) init false;
v_p : \( \mathcal{P} \) init p;
state_p : (sleep, leader, lost) init sleep;

begin if \( p \) is initiator then
    begin \( wsp := true \);
        forall \( q \in \text{Neigh}_p \) do send \( \langle \text{wakeup} \rangle \) to \( q \)
    end;
    while \( wr_p < \#\text{Neigh}_p \) do
        begin receive \( \langle \text{wakeup} \rangle \); \( wr_p := wr_p + 1 \);
            if not \( wsp \) then
                begin \( wsp := true \);
                    forall \( q \in \text{Neigh}_p \) do send \( \langle \text{wakeup} \rangle \) to \( q \)
                end
        end;

(* Now start the tree algorithm *)

while \( \#\{ q : \neg rec_p[q] \} > 1 \) do
    begin receive \( \langle \text{tok}, r \rangle \) from \( q \); \( rec_p[q] := true \);
        \( v_p := \min(v_p, r) \)
    end;

send \( \langle \text{tok}, v_p \rangle \) to \( q_0 \) with \( \neg rec_p[q_0] \);
receive \( \langle \text{tok}, r \rangle \) from \( q_0 \);
\( v_p := \min(v_p, r) \); (* decide with answer \( v_p \) *)
if \( v_p = p \) then \( state_p := \text{leader} \) else \( state_p := \text{lost} \);
forall \( q \in \text{Neigh}_p, q \neq q_0 \) do send \( \langle \text{tok}, v_p \rangle \) to \( q \)
end
\end{verbatim}

Algorithm 7.1 ELECTION ALGORITHM FOR TREES.
Phase alg. and Finn’s alg.

- **Phase algorithm:**
  - compute the smallest identity in a single wave
  - distribute result
  - complexity $O(D|E|)$

- **Finn’s algorithm:**
  - same technique, but uses longer messages
  - complexity $O(N|E|) \times O(N)$
Ring Networks

Unidirected

LeLann 77, $O(N^2)$
Chang-Roberts 79 $\Omega(N^2), \Theta(N \log N)$

Bidirected

Hirshberg et al. 80 $\Omega(N \log N)$
Petersen/Dolev et al. 82 $\Omega(N \log N)$
LeLann

- Idea: each node collects a list of initiator identities and then selects the smallest as leader. Terminate when my own id comes back.
- Assumption: FIFO
- Algorithm for *initiator*:
  - send my pid
  - while not leader or loser
    - if receive other pid *q* then memorize *q* and send *q*
    - if receive my pid
      - then if my pid = min pid then leader else lost
- Algorithm for *non-initiator*:
  - if receive *q* then send *q*

- Chang-Roberts alg. improves complexity by removing obviously losing pids, i.e. pids > current min pid
var \( \text{List}_p \) : set of \( \mathcal{P} \) init \{p\} ;
\( \text{state}_p \);

begin if \( p \) is initiator then
    begin \( \text{state}_p := \text{cand} \); send \( \langle \text{tok}, p \rangle \) to \( \text{Next}_p \); receive \( \langle \text{tok}, q \rangle \);
        while \( q \neq p \) do
            begin \( \text{List}_p := \text{List}_p \cup \{q\} \);
                send \( \langle \text{tok}, q \rangle \) to \( \text{Next}_p \); receive \( \langle \text{tok}, q \rangle \)
            end
            if \( p = \min(\text{List}_p) \) then \( \text{state}_p := \text{leader} \)
                else \( \text{state}_p := \text{lost} \)
    end
else repeat receive \( \langle \text{tok}, q \rangle \); send \( \langle \text{tok}, q \rangle \) to \( \text{Next}_p \);
        if \( \text{state}_p = \text{sleep} \) then \( \text{state}_p := \text{lost} \)
        until false
end

Algorithm 7.2 LeLann’s election algorithm.
\textbf{Algorithm 7.3} \textsc{The Chang–Roberts election algorithm.}
Franklin

- All active identities compare themselves with the closest active neighbour to the left and to the right.
- The identity remains active if it is the local min.
Processes pass around tokens called *identities* (≠ *pids*).

**Problem:**

\[ a \rightarrow b \rightarrow c \rightarrow d \rightarrow a \]

\[ b \text{ has to be compared with } a \text{ and } c, \text{ but the edges are uni-directed} \]

**Solution:**

processes with *active ids* give them to the next proc. with active id

\[ a \rightarrow b \rightarrow c \rightarrow d \rightarrow a \]

then they inform their neighbour of their new id
```latex
\begin{verbatim}
var \textit{ci}_p : \mathcal{P} \quad \text{init } p; \quad \text{(* Current identity of } p \) \\
\textit{acn}_p : \mathcal{P} \quad \text{init undef; \quad (* Id of anticlockwise active neighbor *)} \\
\textit{win}_p : \mathcal{P} \quad \text{init undef; \quad (* Id of winner *)} \\
\textit{state}_p : \{\text{active, passive, leader, lost}\} \quad \text{init active;}

begin
  if \( p \) \text{ is initiator then } \textit{state}_p := \text{active} \quad \text{else } \textit{state}_p := \text{passive};
  \text{while } \textit{win}_p = \text{undef} \text{ do}
    \text{begin if } \textit{state}_p = \text{active then}
      \text{begin send } (\text{one, } \textit{ci}_p) \text{; receive } (\text{one, } q); \textit{acn}_p := q; \\
      \text{if } \textit{acn}_p = \textit{ci}_p \text{ then (* } \textit{acn}_p \text{ is the minimum *)}
        \text{begin send } (\text{smal, } \textit{acn}_p); \textit{win}_p := \textit{acn}_p;
        \text{receive } (\text{smal, } q)
      \text{end}
      \text{else (* } \textit{acn}_p \text{ is current id of neighbor *)}
        \text{begin send } (\text{two, } \textit{acn}_p); \text{receive } (\text{two, } q);
        \text{if } \textit{acn}_p < \textit{ci}_p \text{ and } \textit{acn}_p < q
          \text{then } \textit{ci}_p := \textit{acn}_p
        \text{else } \textit{state}_p := \text{passive}
      \text{end}
    \text{end}
    \text{else (* } \textit{state}_p = \text{passive *)}
      \text{begin receive } (\text{uno, } q); \text{send } (\text{uno, } q);
      \text{receive } m; \text{send } m;
      \text{(* } m \text{ is either } (\text{two, } q) \text{ or } (\text{smal, } q) \text{ *)}
      \text{if } m \text{ is a } (\text{smal, } q) \text{ message then } \textit{win}_p := q.
    \text{end}
  \text{end};
  \text{if } p = \textit{win}_p \text{ then } \textit{state}_p := \text{leader} \text{ else } \textit{state}_p := \text{lost}
end
\end{verbatim}
```

Algorithm 7.7 The Peterson/Dolev-Klawe-Rodeh algorithm.
Arbitrary networks

- unknown topology (for the processes)
- no neighbour knowledge (doesn’t know the pid of the neighbours)

Th. 7.15: Any comparison election alg. for arbitrary networks has worst case and average complexity of at least $\Omega(N \log N + |E|)$
Centralized wave alg. + Extinction ⊂ Election alg.

- Each initiator starts a wave, tags it with its pid $p$
- Each node participates in at most one wave $caw$ at a time
- Ignore message with tag $q > caw$
- For messages with $q = caw$ follow the wave alg.
- If a message with $q < caw$ arrives then reset the wave alg. and set $caw \leftarrow q$
- The node that makes the decision is elected
var \( \text{caw}_p \) : \( \mathcal{P} \)  
i init udef;  (* Currently active wave *)  
rec_p : integer  
i init 0;  (* Number of \((\text{tok}, \text{caw}_p)\) received *)  
father_p : \( \mathcal{P} \)  
i init udef;  (* Father in wave \( \text{caw}_p \) *)  
lrec_p : integer  
i init 0;  (* Number of \((\text{ldr}, \_\)\) received *)  
win_p : \( \mathcal{P} \)  
i init udef;  (* Identity of leader *)

begin if \( p \) is initiator then 
begin \( \text{caw}_p := p \);  
forall \( q \in \text{Neigh}_p \) do send \((\text{tok}, p)\) to \( q \)  
end;  
while \( lrec_p < \#\text{Neigh}_p \) do 
begin receive \text{msg} from \( q \);  
if \text{msg} = \((\text{ldr}, r)\) then 
begin if \( lrec_p = 0 \) then 
forall \( q \in \text{Neigh}_p \) do send \((\text{ldr}, r)\) to \( q \)  
lrec_p := lrec_p + 1; win_p := r  
end  
else (* a \((\text{tok}, r)\) message *)  
begin if \( r < \text{caw}_p \) then (* Reinitialize algorithm *)  
begin \( \text{caw}_p := r \); \( \text{rec}_p := 0 \); \( \text{father}_p := q \);  
forall \( s \in \text{Neigh}_p, s \neq q \) do send \((\text{tok}, r)\) to \( s \)  
end;  
if \( r = \text{caw}_p \) then 
begin \( \text{rec}_p := \text{rec}_p + 1 \);  
if \( \text{rec}_p = \#\text{Neigh}_p \) then  
if \( \text{caw}_p = p \) then forall \( s \in \text{Neigh}_p \) do send \((\text{ldr}, p)\) to \( s \)  
else send \((\text{tok}, \text{caw}_p)\) to \( \text{father}_p \)  
end  
(* If \( r > \text{caw}_p \), the message is ignored *)
end;  
if \( \text{win}_p = p \) then state_p := leader else state_p := lost
end

Algorithm 7.9 Extinction Applied to the Echo Algorithm.
Complexity

If wave alg. A has complexity M then Ex(A) has complexity NM

Extinction + Ring algorithm = Chang-Roberts
Elections and Spanning Trees

- are closely related problems
- $C_E \leq C_T + O(N)$ (spanning tree+TREE)
- $C_T \leq C_E + 2|E|$ (leader+Echo alg.)
- $C_E \leq \Omega(N \log N + |E|)$ (Th. 7.15)

$\Rightarrow$ both $C_E$ and $C_T$ require $\Omega(N \log N + |E|)$ messages
Gallager-Humblet-Spira Minimal Spanning Tree

• Assumption:
  – unique weights $w(e)$ for all edges

•Defs:
  – **Fragment** $F$ is a subtree of the MST of $G$
  – **Outgoing edge** $e$ from $F$ if one of the edge’s nodes is in $F$ and the other is not in $F$.
  – **Core node** is the node with the least weight outgoing edge
GHS (continued)

- Prop. 7.19: If $F$ is a fragment and $e$ is the least weight outgoing edge of $F$, then $F \cup \{e\}$ is a fragment.

Algorithm

Start with single node fragments and incrementally enlarge them using 7.19
Global description of GHS

• Maintain for G = (V,E) a set of fragments such that \( \bigcup_i \text{nodes}(F_i) = V \) and \( i \neq j \iff F_i \cap F_j = \emptyset \)

• start with one-node fragments

• nodes in a fragment cooperate to find the lowest weight outgoing edge (core edge)

• when core edge is found, combine with the other fragment

• Terminate when only one fragment remains
GHS

• How to determine if an edge is outgoing?
  Solution: Fragment names

• How to determine names for combined fragments?
  Solution: The combined fragment gets the name of the larger fragment
GHS

- How to determine the size of a fragment?
  Solution: The number of nodes
  Problem: all nodes have to be updated...
  Solution GHS: use levels

- Initially \( L_i = 0 \),
- \( \text{level}(F_1) > \text{level}(F_2) \) \( \Rightarrow \) \( \text{level}(F_1 \cup F_2) = \text{level}(F_1) \)
- \( \text{level}(F_1) = \text{level}(F_2) \) and \( e_1 = e_2 \) \( \Rightarrow \)
  \( \text{level}(F_1 \cup F_2) = \text{level}(F_i) + 1 \)

Note: A process can change name at most \( \log N \) times
Local description of GHS

• Each node $p$ stores
  – the state of its edges $e$, $stach_p[e] \in \{\text{basic, branch, reject}\}$
  – name of its fragment
  – level of its fragment
  – best weight of outgoing edges from its fragment
  – father channel (i.e. route towards the core node)
GHS

\[ F_1: L_1 = 2, e_1 = ab \]

\[ F_2: L_2 = 1, e_2 = ab \]

\[ F_3: L_3 = 1, e_3 = cd \]

a sends \(<\text{connect,}2>\) to b

b sends \(<\text{connect,}1>\) to a

d sends \(<\text{connect,}1>\) to c

a sends \(<\text{initiate,}2,F_1,\text{state}>\) to b

b forwards \(<\text{init...}>\) on all branch channels except father channel
GHS local description

- on <connect> from lower level or equal level with same core edge => send <initialize>
- on <initialize>, set parent to sender, send initialize to other branches, test basic channels for best channel that accepts.
- when an <accept> and reports from all branches are received then send <report> to father and memorize best weight and best channel
- if report received from a parent then the node is a core node. if the reported weight > best weight then send <changeroot> to best channel. This tells “my” best node to send a <connect> to the other fragment
GHS example

\[\text{parent=z, bestwt=8}\]

\[\text{parent=x, bestwt=∞}\]

\[\text{b is also in F}_2\]
a

<accept>

5<9, so bestch=xz
bestwt=5
changeroot

<report, 5>

5<9, so changerooot

<changeroot>

5<∞, so bestch=ac
bestwt=5, state=found

<report, ∞>

∞>5, so no change of bestch or bestwt

report, 5>

bestwt=5
changeroot

<changeroot>

<report, 5>

bestch=ax
bestwt=5

<connect, L=1>

<report, 5>

bestch=ac
bestwt=5, state=found

<report, ∞>

∞>5, so no change of bestch or bestwt

reports from all branches received

<changeroot>

<accept>
Make GHS an election alg.

- The two last core nodes send each other their pids
- The node with the lowest pid wins.
- Cost: 2 messages
\begin{verbatim}
var state_p : (sleep, find, found) ;
stash_p[q] : (basic, branch, reject) for each q \in Neigh_p ;
name_p, bestw_t_p : real ;
level_p : integer ;
testch_p, bestch_p, father_p : Neigh_p ;
rec_p : integer ;

(1) As the first action of each process, the algorithm must be initialized:
begin  let pq be the channel of p with smallest weight ;
\quad stash_p[q] := branch ; level_p := 0 ;
\quad state_p := found ; rec_p := 0 ;
\quad send (connect, 0) to q
end

(2) Upon receipt of \{ connect, L \} from q:
begin  if L < level_p then (* Combine with Rule A *)
\quad begin  stash_p[q] := branch ;
\quad send (initiate, level_p, name_p, state_p) to q
\quad end
\quad else if stash_p[q] = basic
\quad then (* Rule C *) process the message later
\quad else (* Rule B *) send (initiate, level_p + 1, \omega(pq), find) to q
end

(3) Upon receipt of \{ initiate, L, F, S \} from q:
begin  level_p := L ; name_p := F ; state_p := S ; father_p := q ;
\quad bestch_p := undef ; bestw_t_p := \infty ;
\quad forall r \in Neigh_p : stash_p[r] = branch \land r \neq q do
\quad \quad send (initiate, L, F, S) to r ;
\quad if state_p = find then begin rec_p := 0 ; test end
end
\end{verbatim}

Algorithm 7.10 The Gallager-Humblet-Spira algorithm (Part 1).
(4) procedure test:
    begin if $\exists q \in \text{Neigh}_p : stach_p[q] = \text{basic}$ then
        begin $testch_p := q$ with $stach_p[q] = \text{basic}$ and $\omega(pq)$ minimal;
            send $\langle test, level_p, name_p \rangle$ to $testch_p$
        end
        else begin $testch_p := udef$; report end
    end

(5) Upon receipt of $\langle test, L, F \rangle$ from $q$:
    begin if $L > level_p$ then       (* Answer must wait! *)
        process the message later
    else if $F = name_p$ then (* internal edge *)
        begin if $stach_p[q] = \text{basic}$ then $stach_p[q] := \text{reject}$;
            if $q \neq testch_p$
                then send $\langle \text{reject} \rangle$ to $q$
            else $test$
        end
        else send $\langle \text{accept} \rangle$ to $q$
    end

(6) Upon receipt of $\langle \text{accept} \rangle$ from $q$:
    begin $testch_p := udef$;
        if $\omega(pq) < bestwt_p$
            then begin $bestwt_p := \omega(pq)$; $bestch_p := q$ end;
        report
    end

(7) Upon receipt of $\langle \text{reject} \rangle$ from $q$:
    begin if $stach_p[q] = \text{basic}$ then $stach_p[q] := \text{reject}$;
        $test$
    end

(8) procedure report:
    begin if rec_p = # \{ q : stach_p[q] = branch \land q \neq father_p \} 
        and testch_p = udef then 
        begin state_p := found ; send \{ report, bestwt_p \} to father_p end 
    end

(9) Upon receipt of \{ report, \omega \} from q:
    begin if q \neq father_p 
        then (* reply for initiate message *) 
            begin if \omega < bestwt_p then 
                begin bestwt_p := \omega ; bestch_p := q end ; 
                rec_p := rec_p + 1 ; report 
            end 
        else (* pq is the core edge *) 
            if state_p = find 
                then process this message later 
            else if \omega > bestwt_p 
                then changeroot 
            else if \omega = bestwt_p = \infty then stop 
    end

(10) procedure changeroot:
    begin if stach_p[bestch_p] = branch 
        then send \{ changeroot \} to bestch_p 
        else begin send \{ connect, level_p \} to bestch_p ; 
                stach_p[bestch_p] := branch 
        end
    end

(11) Upon receipt of \{ changeroot \}:
    begin changeroot end

---

\textbf{Algorithm 7.13 The Korach-Kutten-Moran algorithm.}
Algorithm 6.9. Finn's Algorithm.

initialize $\ {\text{Init}} \ (p)$

\begin{verbatim}
begin if $p$ is initiator then
    for all $r \in \text{Out}_p$ do send (sets, Inc$_p$, $N$Inc$_p$) to $r$;
end

while $\text{Inc}_p \neq N$Inc$_p$ do
    begin receive (sets, Inc$_p$, $N$Inc$_p$) from $q$;
        if $q \notin \text{Out}_p$ then $\text{Inc}_p := \text{Inc}_p \cup \text{Inc}_p$;
        if $q \notin \text{Out}_p$ then $\text{NInc}_p := \text{NInc}_p \cup \text{NInc}_p$;
        for all $r \in \text{Out}_p$ do send (sets, Inc$_p$, $N$Inc$_p$) to $r$;
    end
end

end

end
\end{verbatim}
\textbf{Algorithm 6.7} \textsc{The Phase Algorithm}.

\begin{verbatim}
cons \(D\) : integer = the network diameter;

\textbf{var} \(R_{cp}[q]\) : 0..\(D\) init 0, for each \(q \in In_p\);
\((\ast \text{ Number of messages received from } q \ast)\)

\(S_{tep}\) : 0..\(D\) init 0;
\((\ast \text{ Number of messages sent to each out-neighbor } \ast)\)

begin if \(p\) is initiator then
  begin forall \(r \in Out_p\) do send \((\text{tok})\) to \(r\);
    \(S_{tep} := S_{tep} + 1\)
  end;

  while \(\min_q R_{cp}[q] < D\) do
    begin receive \((\text{tok})\) (from neighbor \(q_0\));
      \(R_{cp}[q_0] := R_{cp}[q_0] + 1\);
      if \(\min_q R_{cp}[q] \geq S_{tep}\) and \(S_{tep} < D\) then
        begin forall \(r \in Out_p\) do send \((\text{tok})\) to \(r\);
          \(S_{tep} := S_{tep} + 1\)
        end
    end;

end
\end{verbatim}