Election algorithms

- Definition of election algorithm, assumptions
- election algorithm on a tree
- rings
  - LeLann
  - Chang and Roberts
  - Dolev-Klawe-Rodeh
- arbitrary networks - extinction algorithm

What is election algorithm

- Election algorithm satisfies the following properties:
  - (uniformity) each process has the same local algorithm
  - (decentralization) a computation can be initiated by arbitrary subset of processes
  - (termination) each computation terminates
  - (safety) in every terminal state there is one and only one process in the state of leader
- Assumptions
  - system is fully asynchronous - not necessary but useful
  - each process has unique identity (name), process identities are totally ordered (can be compared) - needed to break the symmetry, smallest identity wins (becomes leader)
  - the only operation allowed on identities is comparison - in asynchronous systems arbitrary (non-comparison algorithms can do no better)
  - each message may contain up to a constant number of identities - so complexities of algorithms can be compared

Waves on trees and election

```
var ws_p    : boolean                      init false ;
    wr_p    : integer                      init 0 ;
    rec_p[q] : boolean for each q ∈ Neigh_p init false ;
    v_p     : 𝒫                            init p ;
    state_p : (sleep, leader, lost)        init sleep ;

begin if p is initiator then
        begin ws_p := true ;
              forall q ∈ Neigh_p do send ⟨wakeup⟩ to q
        end ;
      while wr_p < #Neigh_p do
        begin receive ⟨wakeup⟩ ; wr_p := wr_p + 1 ;
              if not ws_p then
                begin ws_p := true ;
                      forall q ∈ Neigh_p do send ⟨wakeup⟩ to q
                end
        end ;
      (* Now start the tree algorithm *)
      while #{q : ¬rec_p[q]} > 1 do
        begin receive ⟨tok, r⟩ from q ; rec_p[q] := true ;
              v_p := min(v_p, r)
        end ;
      send ⟨tok, v_p⟩ to q_0 with ¬rec_p[q_0] ;
      receive ⟨tok, r⟩ from q_0 ;
      v_p := min(v_p, r) ; (* decide with answer v_p *)
      if v_p = p then state_p := leader else state_p := lost ;
      forall q ∈ Neigh_p, q ≠ q_0 do send ⟨tok, v_p⟩ to q
end
```

- Use wave tree algorithm
- first phase - "wake up" all processes (non-initiators)
  - a wakeup message is propagated from initiators to other processes
- second phase - do a wave on the tree
  - leaves start
  - id of lowest process in subtree is attached to token
- message complexity: 4*N*-4 O(*N*) (two wakeup messages and two tokens are sent along each channel)
- time complexity - 3*D*+1 (*D* - diameter) - O(*N*)
  - *D* - to send wakeup messages, in *D*+1 tree algorithm starts *D* - to make first decision, *D* - to propagate decision

LeLann's algorithm

```
var List_p   : set of 𝒫    init {p} ;
    state_p ;

begin if p is initiator then
        begin state_p := cand ; send ⟨tok, p⟩ to Next_p ; receive ⟨tok, q⟩ ;
              while q ≠ p do
                begin List_p := List_p ∪ {q} ;
                      send ⟨tok, q⟩ to Next_p ; receive ⟨tok, q⟩
                end ;
              if p = min(List_p) then state_p := leader
                                 else state_p := lost
        end
      else repeat receive ⟨tok, q⟩ ; send ⟨tok, q⟩ to Next_p ;
                  if state_p = sleep then state_p := lost
           until false
end
```

- Each initiator sends token with its id around the ring
- a process forwards foreign token and records the identity it carries
- a process decides when it receives its own token
- if processes gets a message before it sends its own - it loses
- complexity
  - message - O(N^2)
  - time - O(*N*)

Chang and Roberts's algorithm (CR)

```
var state_p ;

begin if p is initiator then
        begin state_p := cand ; send ⟨tok, p⟩ to Next_p ;
              repeat receive ⟨tok, q⟩ ;
                     if q = p then state_p := leader
                     else if q < p then
                       begin if state_p = cand then state_p := lost ;
                             send ⟨tok, q⟩ to Next_p
                       end
              until state_p = leader
        end
      else repeat receive ⟨tok, q⟩ ; send ⟨tok, q⟩ to Next_p ;
                  if state_p = sleep then state_p := lost
           until false
end
(* Only the leader terminates the program. It floods a message to all
processes to inform them of the leader's identity and to terminate *)
```

- Process propagates foreign message only when the original sender may become a leader
- complexity
  - time - same as LeLanns'
  - message
    - worst case - same as LeLann's
    - average - O(*N* log *N*)

Dolev-Klawe-Rodeh algorithm (DKR)

- Worst case message complexity of Chang-Roberts is still O(N^2) -- in the worst case the identities that are not leaders are allowed to propagate
- idea of DKR - eliminate wrong identities as soon as possible
- active/passive processes as in CR
  - active - propagates its current identity (stored in variable *ci*) to downstream processes
  - passive - forwards messages
- algorithm proceeds in rounds, each round has two phases:
  - propagation - each active process sends message <*one, ci*> to downstream active process, received id stored in *acn*
  - elimination - each active process sends a message <*two, acn*> to downstream neighbor when <*two*,.> arrives, the receiver compares it with its can thus the following identity "catches up" with preceding
- when process gets his own identity - its is a winner winner sends message <*smal,id*> informing others

DKR

```
var ci_p   : P   init p ;      (* Current identity of p *)
    acn_p  : P   init udef ;   (* Id of anticlockwise active neighbor *)
    win_p  : P   init udef ;   (* Id of winner *)
    state_p: (active, passive, leader, lost) init active ;

begin if p is initiator then state_p := active else state_p := passive ;
      while win_p = udef do
        begin if state_p = active then
                begin send ⟨one, ci_p⟩ ; receive ⟨one, q⟩ ; acn_p := q ;
                      if acn_p = ci_p then (* acn_p is the minimum *)
                        begin send ⟨smal, acn_p⟩ ; win_p := acn_p ;
                              receive ⟨smal, q⟩
                        end
                      else (* acn_p is current id of neighbor *)
                        begin send ⟨two, acn_p⟩ ; receive ⟨two, q⟩ ;
                              if acn_p < ci_p and acn_p < q
                                then ci_p := acn_p
                                else state_p := passive
                        end
                end
              else (* state_p = passive *)
                begin receive ⟨one, q⟩ ; send ⟨one, q⟩ ;
                      receive m ; send m ;
                      (* m is either ⟨two, q⟩ or ⟨smal, q⟩ *)
                      if m is a ⟨smal, q⟩ message then win_p := q
                end
        end ;
      if p = win_p then state_p := leader else state_p := lost
end
```

DKR message complexity estimate

- Claim: if there were k identities at the beginning of the round - only $k/2$ survive after the round
  - each process (it it survives round) assumes the identity of the first upstream neighbor
  - out of two neighbor identities only one survives the round
- there can be at most log$N$ rounds, $2N$ messages are exchanged during round, algorithm completes with N messages informing processes of results
  - thus message complexity of the algorithm is:

    $2N \log N + N = O(N \log N)$

Election on arbitrary networks, extinction

- Election can be done on networks of arbitrary topology by using any centralized wave algorithm.
- The technique is called *extinction.*
- each initiator starts a separate wave. To distinguish waves, initiator's id is attached to token.
- Only the wave of the initiator with the lowest id is allowed to finish
- A process stores the wave with the lowest id that it saw in currently active wave (*caw*) variable. The process ignores tokens from waves with higher id than *caw*
- If a process receives a token from a wave with still lower id than *caw,* the process copies that id into *caw*
- complexity
  - message NM where M - number of messages used in the base wave algorithm
  - time NT where T - time the wave takes

Extinction applied to Echo

```
var caw_p    : P        init udef ;  (* Currently active wave *)
    rec_p    : integer  init 0 ;     (* Number of ⟨tok, caw_p⟩ received *)
    father_p : P        init udef ;  (* Father in wave caw_p *)
    lrec_p   : integer  init 0 ;     (* Number of ⟨ldr, .⟩ received *)
    win_p    : P        init udef ;  (* Identity of leader *)

begin if p is initiator then
        begin caw_p := p ;
              forall q ∈ Neigh_p do send ⟨tok, p⟩ to q
        end ;
      while lrec_p < #Neigh_p do
        begin receive msg from q ;
              if msg = ⟨ldr, r⟩ then
                begin if lrec_p = 0 then
                        forall q ∈ Neigh_p do send ⟨ldr, r⟩ to q ;
                      lrec_p := lrec_p + 1 ; win_p := r
                end
              else (* a ⟨tok, r⟩ message *)
                begin if r < caw_p then (* Reinitialize algorithm *)
                        begin caw_p := r ; rec_p := 0 ; father_p := q ;
                              forall s ∈ Neigh_p, s ≠ q
                                  do send ⟨tok, r⟩ to s
                        end ;
                      if r = caw_p then
                        begin rec_p := rec_p + 1 ;
                              if rec_p = #Neigh_p then
                                if caw_p = p
                                  then forall s ∈ Neigh_p
                                         do send ⟨ldr, p⟩ to s
                                  else send ⟨tok, caw_p⟩ to father_p
                        end
                      (* If r > caw_p, the message is ignored *)
                end
        end ;
      if win_p = p then state_p := leader else state_p := lost
end
```