Leader Election

CS60002: Distributed Systems

Pallab Dasgupta
Dept. of Computer Sc. & Engg.,
Indian Institute of Technology Kharagpur

Leader Election in Rings

- **Models**
  - Synchronous or Asynchronous
  - Anonymous (no unique id) or Non-anonymous (unique ids)
  - Uniform (no knowledge of *N*, the number of processes) or non-uniform (knows *N*)
- **Known Impossibility Result:**
  - There is no synchronous, non-uniform leader election protocol for anonymous rings

Election in Asynchronous Rings

- **LeLann's and Chang-Robert's Algorithms**
  - send own id to node on left
  - if an id received from right, forward id to left node only if received id greater than own id, else ignore
  - if own id received, declares itself "leader"
- **Works on unidirectional rings**
- **Message complexity = $O(n^2)$**

Hirschberg-Sinclair Algorithm

- Operates in phases, requires bidirectional ring
- In the k^{th} phase, send own id to 2^k processes on both sides of yourself (directly send only to next processes with id and *k* in it)
- If id received, forward if received id greater than own id, else ignore
- Last process in the chain sends a reply to originator if its id less than received id
- Replies are always forwarded
- A process goes to $(k+1)^{th}$ phase only if it receives a reply from both sides in k^{th} phase
- Process receiving its own id – declare itself "leader". At most lg*n* rounds

Features: Hirschberg-Sinclair

- **Message Complexity: $O(n \lg n)$**
- **Lots of other algorithms exist for rings**
- **Lower Bound Result:**
  - **Any *comparison-based* leader election algorithm in a ring requires $\Omega(n \lg n)$ messages**

The Echo Algorithm – a wave algorithm

var rec_p : integer init 0; // Counts no of recvd mesgs
$father_p$: process init *udef*;

For the initiator

```
begin forall q ∈ Neigh_p do send ⟨ tok ⟩ to q ;
        while rec_p < #Neigh_p do
                begin receive ⟨ tok ⟩ ; rec_p = rec_p + 1 end ;
                decide
end
```

For non-initiators

```
begin receive ⟨ tok ⟩ from neighbor q ; father_p = q ; rec_p = rec_p + 1 ;
        forall q ∈ Neigh_p, q ≠ father_p do send ⟨ tok ⟩ to q ;
        while rec_p < #Neigh_p do
                begin receive ⟨ tok ⟩ ; rec_p = rec_p + 1 end ;
        send ⟨ tok ⟩ to father_p
end
```

Extinction on The Echo Algorithm

```
var caw_p     : process   init udef ;  // Currently active wave
    rec_p     : integer   init 0;      // No of ⟨ tok, caw_p ⟩ received
    father_p  : process   init udef;   // Father in wave caw_p
    lrec_p    : integer   init 0;      // No of ⟨ ldr, . ⟩ received
    win_p     : process   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 ;
```

Extinction on Echo Algorithm contd..

```
else // mesg is a ⟨ tok, r ⟩ message
    begin if r < caw_p then  // Reinitialize the algorithm
            begin caw_p = p ; 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 then the message is ignored – extinction
      end
end ;
if win_p = p  then state_p = leader else state_p = lost
end
```

Features

- *If A is a centralized wave algorithm using M messages per wave, the algorithm Ex(A) elects a leader using at most NM messages*