04_03_Election

1.Background

1.1. Goal

- a single process should get the privilege to take some action
- This process has to be elected

1.2. Trivial Solution

- every process sends its id to every other process
- message complexity: n^2
- time complexity: 1

1.3. Anonymous Network

- A network is **anonymous** when the processors do not have ids
- In **anonymous** rings, election is **impossible** (so use randomization to create random ids)

1.4. Comparison-Based and Non-comparison Based

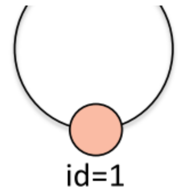
1.4.1. Comparison-Based Algorithm

- sending, receiving, and comparing ids are the only operations allowed
- The message complexity of comparison-based election algorithms in **rings** is of order **n·log(n)**

1.4.2. Non-Comparison-Based Algorithm

Non-comparison-based algorithms in a **synchronous ring** of size n with **positive ids** can be more efficient

- elect process with **minimum id**
- if some process has **id=1**, it sends it in round **1** along the ring
- every process relays this message (in the first **n** rounds)
- if in the first **n** rounds nothing is received and a process has **id=2**, this process sends its id along the ring in round **n+1**
- in general, if in the first **(k-1)n** rounds nothing is received and a process has id=**k**, it sends it in round **(k-1)n+1**
- **time and message complexity: O(n)**



2.Bidirectional Rings

2.1. Hirschberg's and Sinclair's election algorithm in a bidirectional ring

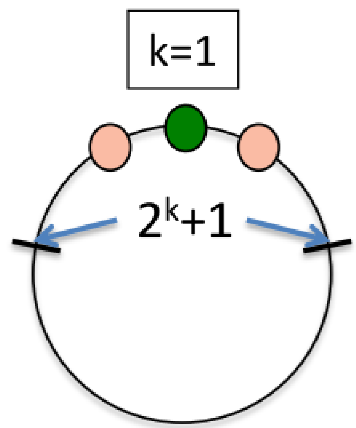
Main Idea

- every process finds out if it has the largest id of ever larger segments of the ring (of size 2^{k+1})

Process

- In round k, every active process **sends** a **PROBE**
  message to each of its neighbors with
  - its id
  - a hop count of 2^{k-1}
- When a process **receives** such a message,
  - if its **own id is larger**, it **discards** it
  - else,

- if the hop count is **positive**, it **decrements** the hop count and sends the message along
- else it sends an **OK message** with the **id** back



- When a process receives **two phase-k OK** messages with its own id, it initiates phase **k+1**
- When a process **does not receive** two phase-k OK messages with its own id, it **does not start** phase k+1
- A process is **elected** when it receives both of its own messages from the **"wrong" side**

2.2. An election algorithm in a bidirectional ring

Procedure

1. First round:
    a. every process **exchanges process ids** with its two neighbors
    b. a process remains **active** if its id is larger than those of its two neighbors
    c. otherwise, it becomes **passive**
2. Every next round:

    repeat the first round in the **virtual ring** consisting of the processes that remain **active**
3. Elected: The process that **receives its own id** is elected

Understanding

It like bubble sort algorithm, n every round: **at least half of** the still active processes become passive

Complexity

Message Complexity: 2n log(n)

3. Unidirectional Rings

3.1. Assumption

the neighbor a processor can send to is its right-hand neighbor

3.2. Chang’s and Roberts’s election algorithm in a unidirectional ring

Assumption

1. The maximum one is chosen

Procedure

1. Every process may **spontaneously start** by sending its id to its neighbor
2. When a process receives an id, it **compares** it with its own id
   a. id=own_id → process has been **elected**
   b. id<own_id → send **own_id** if not already done so
   c. id>own_id → send **id** along

Implementation

Implementation:

I. Spontaneously starting the election

```
id_sent ← true
send(id)
```

II. Receiving a message

```
upon receipt of (nid) do
    if (nid=id) then elected ← true
    if ((nid < id) and (¬id_sent)) then
        id_sent ← true
        send (id)
    if (nid > id) then
        id_sent ← true
        send (nid)
```

> Chang_election_implementation

Complexity

1. Worst case, when the order of the ids is decreasing : $(n+1)n/2$
2. Best case: n
3. Average n log(n)
4. Average: $nlog(n)$

3.4. Peterson’s Election algorithm

Idea

a simulation in a unidirectional ring of algorithm 2.2 in a bidirectional ring

Prepare

1. Process: has an id

Procedure

Every Process:

1. First set **tid**=id
2. sends its **tid** to its (downstream) neighbor
3. receives in variable **ntid** the id of its (upstream) neighbor
4. sends **max(tid,ntid)** to its neighbor
5. receives this value in variable **nntid**
6. if **ntid ≥ tid and ntid ≥ nntid,** it remains **active** and sets **tid=ntid**
7. otherwise turns **passive** (only relays messages in subsequent rounds))

Every subsequent round: the algorithm of round 1 is repeated in the virtual ring of active processes

Elected: The process that **receives its own id** has been elected

Understanding

From the procedure:

1. for three nodes follows the uni-direction, the process next to the process with the largest tid will still be active
2. the largest tid will be transferred to the active process

A process **with a large id will become passive immediately**, but its **id will live on** (in the next active process). But from the Implementation, we will see the correct process will finally be elected when it received its id although it may in passive state.

Implementation

Implementation:

I. Active processes

```
tid ← id
do forever
    send (tid); receive (ntid)
    if (ntid=id) then elected ← true
    send (max(tid,ntid)); receive (nntid)
    if (nntid=id) then elected ← true
    if ((ntid>=tid) and (ntid>=nntid)) then
        tid ← ntid
    else goto relay
```

II. Relay processes

```
relay:
do forever
    receive (tid)
    if (tid=id) then elected ← true
    send (tid)
```

Peterson’s Election Implementation

Discussions

1. for which arrangement of ids along the ring does the algorithm terminate after one round?

Answer: increasing or decreasing order

2. for which arrangement of ids along a ring of size $n = 2k$ does the algorithm use k rounds

Answer: bit-reversal ordering: for example 00 10 01 11

Complexity

message complexity: 2nlog(n)

time complexity: n processes: 2n-1

Property

if the id of process $P_i(id_i)$ still survives in some active process P_j then all processes between P_i and P_j are passive (including P_i)

4. Complex Network

4.1. Straightforward solution

Every process sends its id directly to everybody else: n^2 messages and constant time

4.2. Afek's and Gafni's synchronous alg

Assumption

1. an asynchronous system with a **complete network**
2. In the synchronous model, a **global clock** is connected to all nodes. The time interval between two consecutive pulses of the clock is a **round.**
3. any number of nodes may **spontaneously start** the algorithm in possibly different rounds
4. The algorithm proceeds in **rounds**
5. The total number of process is known by at least candidates

Idea:

cut back on the number of messages(message optimal) by successively sending an id to **ever larger sets of processes** and waiting for an ack from all of them

Complexity

1. message optimal: 2n log(n) messages
2. Number of rounds: 2 log(n)

Prepare

1. Both types of processes keep track of their **level**, which is the number of rounds since their start
2. **Message types:**
   a. Candidate messages: **(level,id)**
   b. acks
3. A node that start this algorithm, **spawns:**
   a. a candidate process (CP)
   b. an ordinary process (OP)
4. A node that awakens due to the reception of a message only spawns an ordinary process

Procedure

1. In every round:
    a. every **ordinary process** :
        i. first increases its **owner-level by one** (so the level is the number of rounds since their start)
        ii. then, inspects the **newly received messages** to update its owner-level and owner-id if necessary: each round, choose the largest one
    b. candidate process:
        i. increase the receiving set $K = \min(2^{level/2}, |E|)$
        ii. increasing level
        iii. send message to receiving set
        iv. a candidate process that does not receive all acks it expects in a certain round, is **killed**
2. First round:
    a. every ordinary process:
    b. set owner to itself
    c. if receive must change
    d. every candidate process, set its ordinary part's owner to its own

Implementation

I. The candidate process

```
E ← set of all links connected to the node
level ← -1
do forever
   level ← level + 1
   if (level mod 2 = 0) then
      if (E = ∅) then
         elected ← true
      else
         K ← min(2^(level/2), |E|)
         E′ ← any subset of E of K elements
         send(level,id) on all links in E′
         E ← E \ E′
   else
      A ← set of all acks received
      if (|A| < K) then STOP
```

II. The ordinary process

```
link ← nil
level ← -1
do forever
   send (ack) over link
   level ← level + 1
   R ← set of all candidate messages received
   (nlevel,nid) ← lexicographic maximum in R
   if ((nlevel,nid) > (level,id)) then
      (level,id) ← (nlevel,nid)
      link ← link over which (nlevel,nid) is received
   else
      link ← nil
```

```
unused := { the set of n links incident to the candidate }
level := -1 ;
Each round do:
   level := level + 1 ;
   If level is even
   Then
      If unused is empty
      Then
         ELECTED, STOP
      Else
         E := Minimum ( 2^(level/2), | unused | ) ;
         Send (level, id) over E links from unused, and
         remove these links from unused ;
   Else   /*  level is odd  */
      Receive all acknowledgement type messages
      If received less than E acknowledgements
      Then
         Stop           /*  Not a candidate any more */
End each round.

/* The ordinary process program */

 L* := nil ;
 owner-level := -1 ;
 owner-id := id ;
 Each round do:
    Send an acknowledgement over L* ;
    owner-level := owner-level + 1 ;
    Receive all candidate messages {(level,id) over link L};
    Let (level*, id*) be the lexicographically largest
    ( level, id) candidate message, and
    L* the link over which it arrived ;
    If (level*, id*)>(owner-level, owner-id)
    Then
       (owner-level, owner-id ):= (level*, id*) ;
    Else
       L* := nil ;
 End each round.
```

Figure 1: The Synchronous Algorithm

> Afek and Ggafni algorithm_synch Implementation

> Afek and Ggafni algorithm_synch Implementation_2.png

Note:

1. The id in The ordinary process means the “owner-id”, more specificially, the id of the current owner

Understanding

1. meaning of **ack** = “you are bigger” (the process sending the ack is and is by the sending process)

   captured

   owned

2. a process adopts the largest id it has ever seen (**which is the id of its current owner**)
3. Each candidate tries to capture all other nodes by sending messages on all the links incident to it. The candidate that has succeeded in capturing all its neighbors elects itself as the leader
4. Consider largest process in the earliest start set:
   a. for a single ordinary nodes, if the largest one arrive earliest, it will receive ack;
   b. if arrive later, the increment of owner id in ordinary process will guarantee the largest process will receive ack only if it start in the earliest round;
5. Consider not largest candidate in the earliest start set

   a. it may arrive a node earlier than some node and get ack, but it will definitely have some node that it cannot become owner because the largest candidate has already become owner

6. the advantages of using of level is it always catch the earliest+largest one, but not the largest one that may start later, the speed will be faster

Complexity

1. Time complexity: log(n)
2. messages complexity: 2n log(n)

so the **total number of candidate messages** is at most

$$\sum_{i=1}^{\log n} (n / 2^{i-1}) 2^{i-1} = n \log n$$

4.3. Afek’s and Gafni’s asynchronous alg

Assumption

Preparation

1. candidate message **(level,id)**, the **level** indicates the number of nodes a candidate process has captured
2. ordinary captured processes:
   a. maintain a **pointer toward owner**
   b. a **pointer toward potential-owner** to their current and potential new owner

Mean Process

New mechanism that is not in Afek synchronous alg is specified here:

1. The **level** of a node is now used to indicate the number of nodes it has captured
2. if a candidate process captures a node that had already previously been captured (by anode with a lower (level, id) at the time of capturing), the level of the previous owner is not correct anymore then it will be **killed** by the ordinary process
3. nodes try to capture **one of other nodes** at a time
4. It does not make sense for a node to wait for “all” messages it will receive to select the largest id, so the node will **react as soon as it receives** a single message, different from in synchronize situation, has a “queue” to store the message receive in a single round

When candidate P arrives at node v which is currently owned by candidate Q, the following rule is used:

If $(Level(P), id(P)) < (Level(v), id(Q))$, P is killed.
If $(Level(P), id(P)) > (Level(v), id(Q))$, (1) v gets P's level, and (2) P is sent to Q.

Upon arriving to Q:
If $(Level(P), id(P)) < (Level(Q), id(Q))$, P is killed.
If $(Level(P), id(P)) > (Level(Q), id(Q))$, then (1) Q is killed or Q has been killed already, and (2) P captures v.

Upon returning to its initiating node from a successful capturing, P increases its level by one.

Afek and Ggafni algorithm_Asynch_2

Implementation

```
I. The candidate process
while (untraversed ≠ ∅) do
    link ← any untraversed link
    send (level,id) on link
  R: receive (level′,id′) on link′
    if ((id=id′) and (killed=false)) then
       level ← level+1
       untraversed ← untraversed \ link
    else
       if ((level′,id′) < (level,id)) then goto R
       else
          send (level′,id′) on link′
          killed ← true
          goto R
if (killed = false) then elected ← true

II. The ordinary process
do forever
    receive (level′,id′) on link′
    case (level′,id′) of
       (level′,id′) < (level,owner-id):  ignore
       (level′,id′) > (level,owner-id):
          potential-owner ← link′
          (level,owner-id) ← (level′,id′)
          if (owner=nil) then owner ← potential-owner
          send (level′,id′) on owner-link
       (level′,id′) = (level,owner-id):
          owner ← potential-owner
          send (level′,id′) on owner-link
```

> Afek and Ggafni algorithm_Asynch Implementation.png

Understanding

1. The algorithm will always choose the process that capture faster
2. a candidate who has done more work than another (higher level=captured more nodes), will not be killed by a candidate who has done less work

Complexity

Time complexity is O(n): capther all others

Message complexity is n log(n)

5. General Networks

Main Process

1. an initiator **create a spanning tree** in the network & **propagating his id**
    a. if id is the largest, leaves propagate back a success message
    b. if not the largest, larger ids nodes ignore and not propagate
2. if the initiator has received success message on all its links, **elected**
3. else other nodes become initiator in turn