Election

- The problem
- Bidirectional rings
- Unidirectional rings
- Complete networks

Distributed Algorithms (IN4150)

D.H.J. Epema

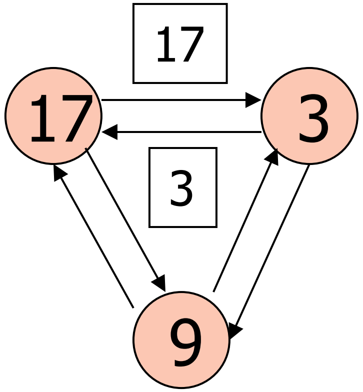
2013-2014

Parallel and Distributed Group

Delft University of Technology

Election (1/3): the problem

- **Problem**: a **single process** should get the privilege to take some action
- This process has to be **elected**
- Usually modeled in a DS in which processes have unique (integer) ids as electing **the process with the largest id**
- Then: election=**maximum/extrema finding**
- **Trivial solution:**
  - every process sends its id to every other process
  - message complexity: $\mathbf{n^2}$
  - time complexity: 1
- So the challenge is to devise **efficient** algorithms



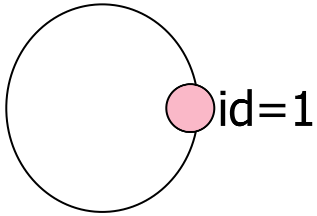
TU Delft

Election (2/3)

- Election has in particular been studied in
  - unidirectional rings
  - bidirectional rings
  - complete networks
- A network is **anonymous** when the processors do not have ids
- In anonymous rings, election is **impossible** (so use **randomization** to create random ids)
- **Comparison-based algorithms:** 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)**

TU Delft

Election (3/3)

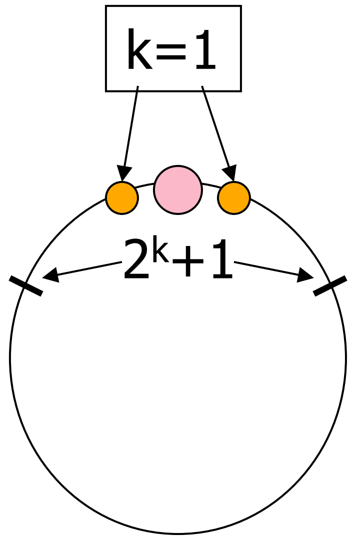


- **Non-comparison-based** algorithms in a **synchronous** ring of size **n** with positive ids can be more efficient:
  - elect process with **minimum id**
  - if some process has **id=1**, it sends it in round **1** along the ring
  - every process relays this message (in the first **n** rounds)
  - 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)**
- From now on: **comparison-based**

TU Delft

Bidirectional ring: solution 1 (1/3)

- **Idea**: every process finds out if it has the largest id of ever larger segments of the ring (of **size 2^k+1**)
- In round **k**, every **active process** sends a message to each of its neighbors with
  - its id
  - a hop count of $\mathbf{2^{k-1}}$
  - a direction "away"
- 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 back (with hop count $\mathbf{2^{k-1}}$)

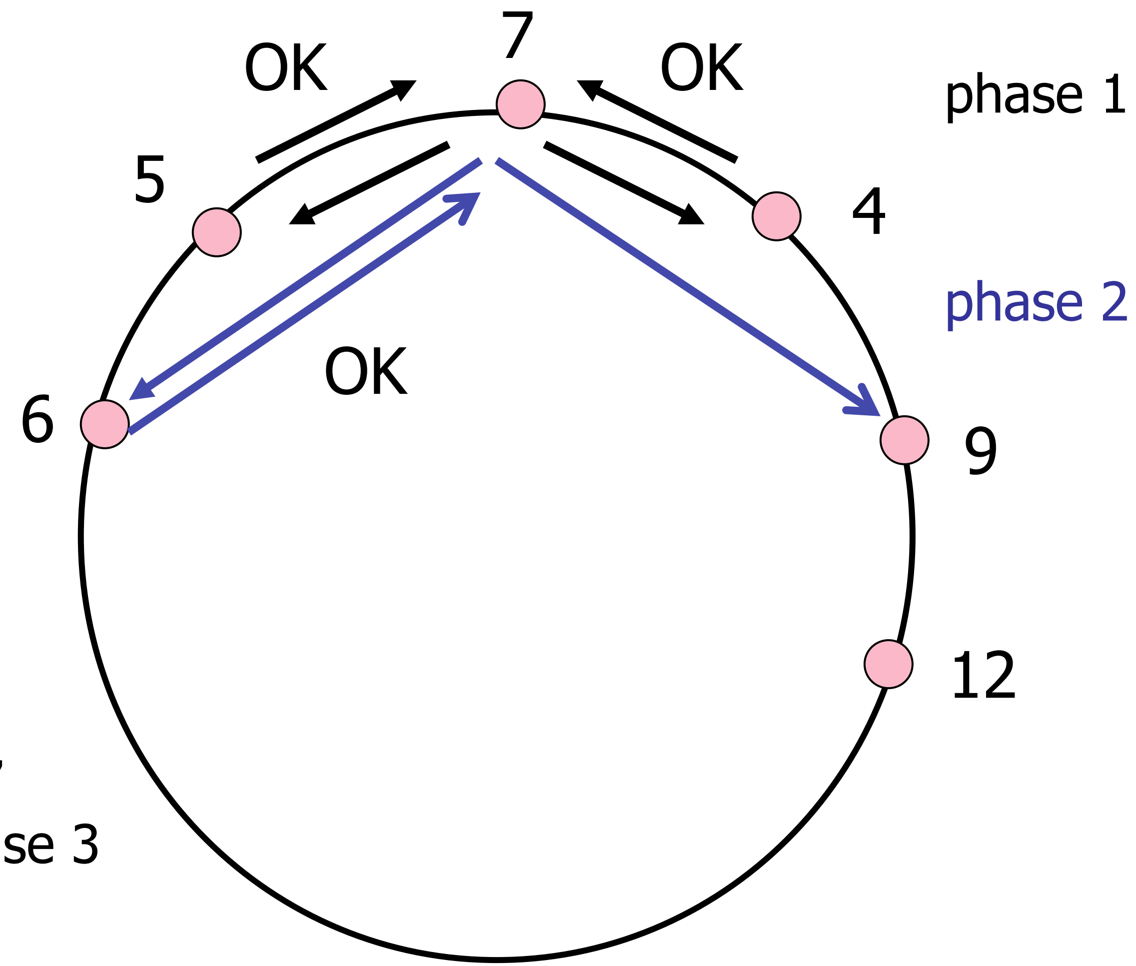


Bidirectional ring: solution 1 (2/3)

- OK messages are returned to the originator
- When a process **receives** two phase-**k** OK messages, it initiates phase **k+1**
- When a process **does not receive** two phase-**k** OK messages, it does not start phase **k+1**
- A **process is elected** when
- Message complexity: **O(n·log(n))**

TU Delft

Bidirectional ring: solution 1 (3/3)



the process with 7
does not start phase 3

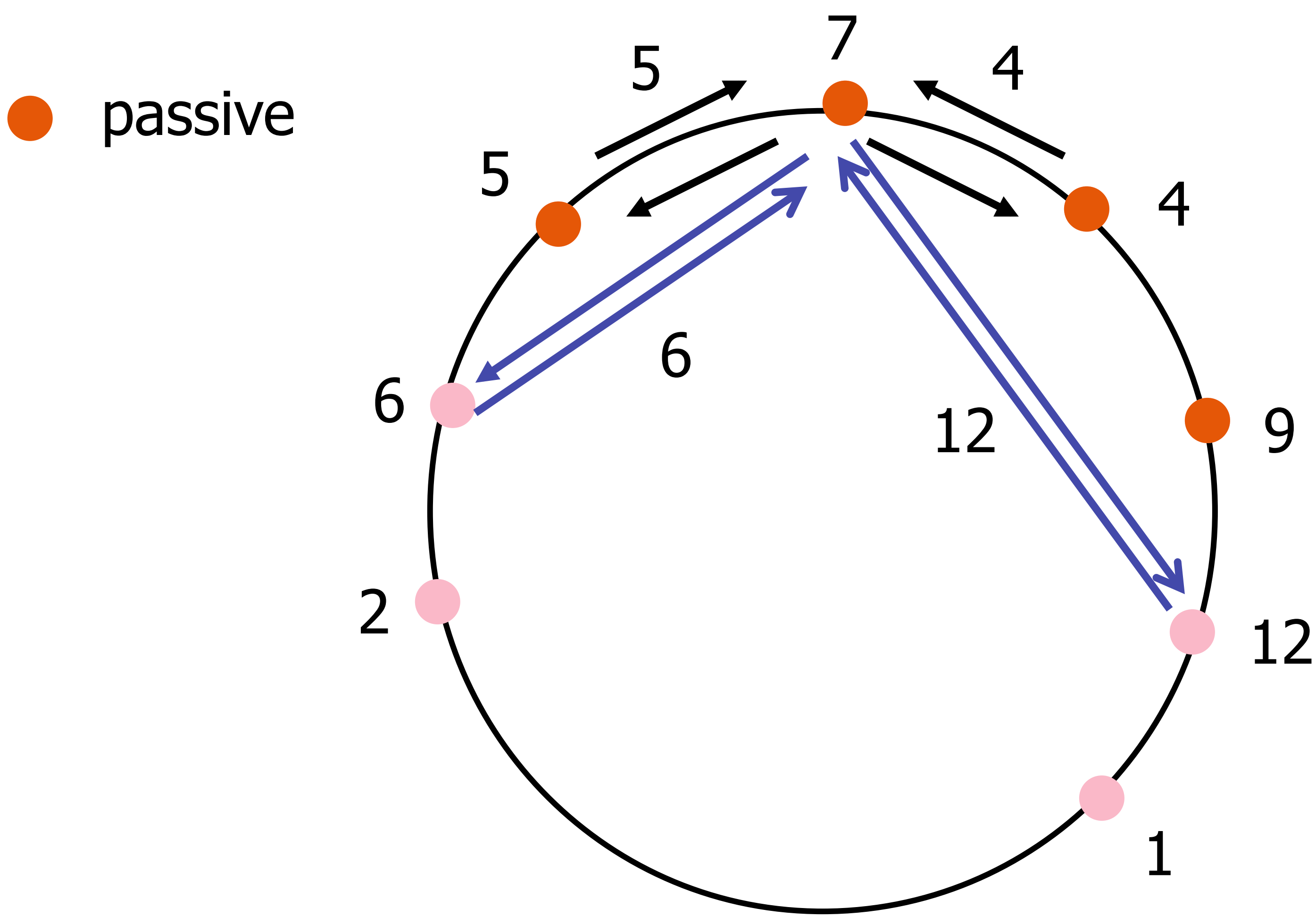
TU Delft

Bidirectional ring: solution 2 (1/2)

- In the **first round**:
  - every process exchanges process ids with its **two neighbors**
  - a process remains **active** if its id is larger than those of its two neighbors
  - otherwise, it becomes **passive**
- **Every next round**: repeat the first round in the **virtual ring** consisting of the processes that remain active
- The process that receives its own id is **elected**
- In every round: **at least half** of the still active processes become passive
- Message complexity: **$2n \cdot \log(n)$**

TU Delft

Bidirectional ring: solution 2 (2/2)



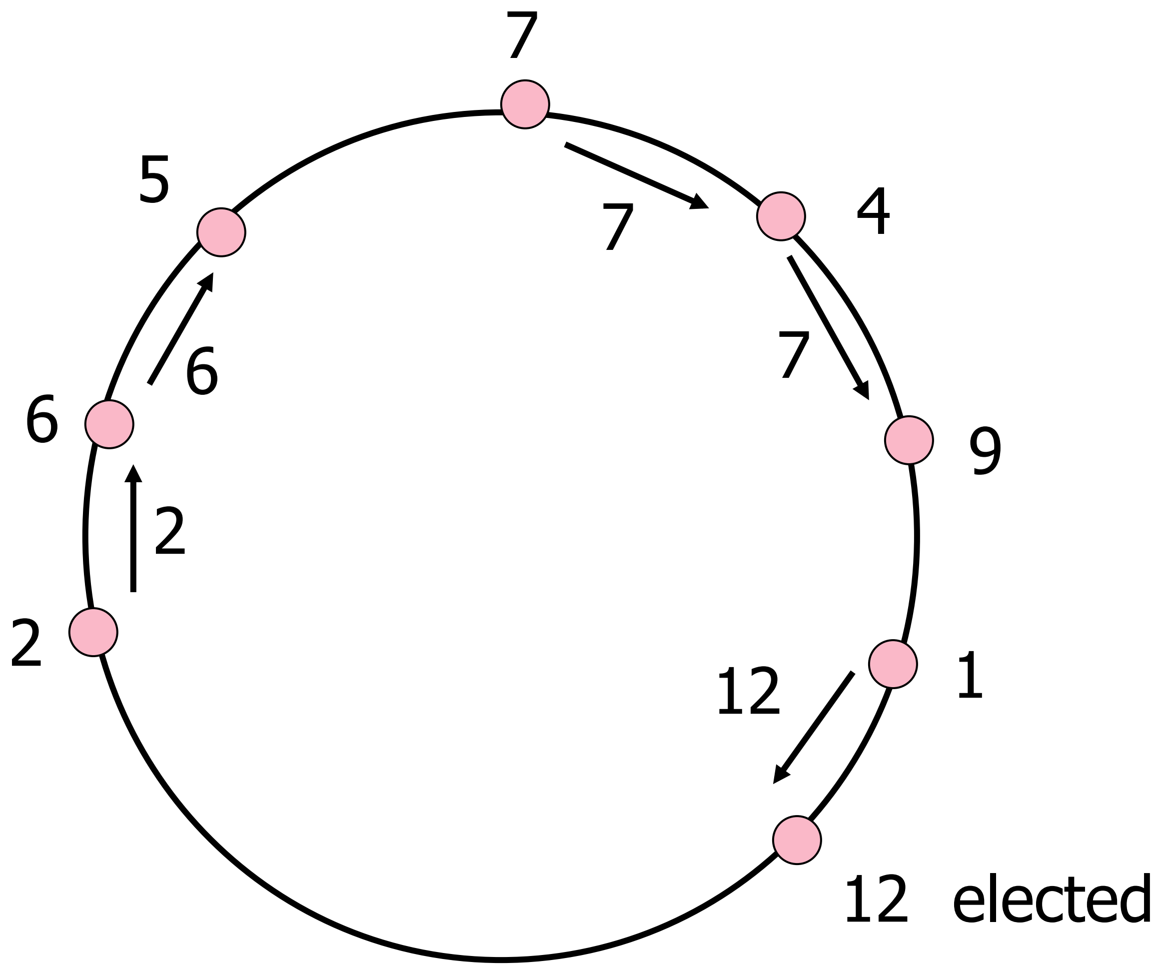
TU Delft

Chang-Roberts algorithm (1/3)

- Is a solution in a **unidirectional ring**
- Every process may spontaneously start by sending its id to its neighbor
- When a process receives an id, it compares it with its own id own_id:

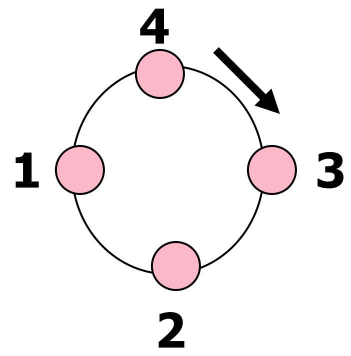
  - **id=own_id** process has been elected
  - **id<own_id** send **own_id** if not already done so
  - **id>own_id** send **id** along

Chang-Roberts algorithm (2/3)



Chang-Roberts algorithm (3/3)

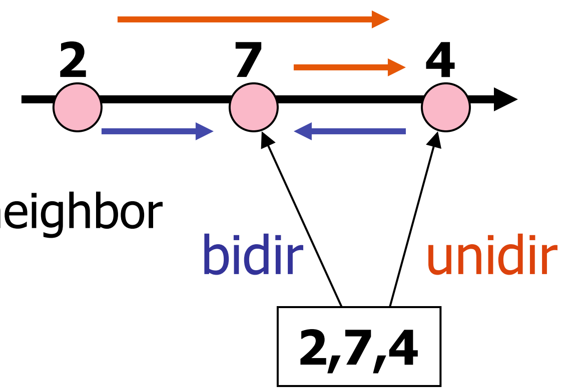
- **Question**: what is the worst-case message complexity?
- **Answer**: $\mathbf{n^2}$
- **Question**: when does the worst case occur?
- **Answer**: when the order of the ids is decreasing, id **i** travels **i** hops (ids 1,2,…,n)



- Average message complexity: **n·log(n)**

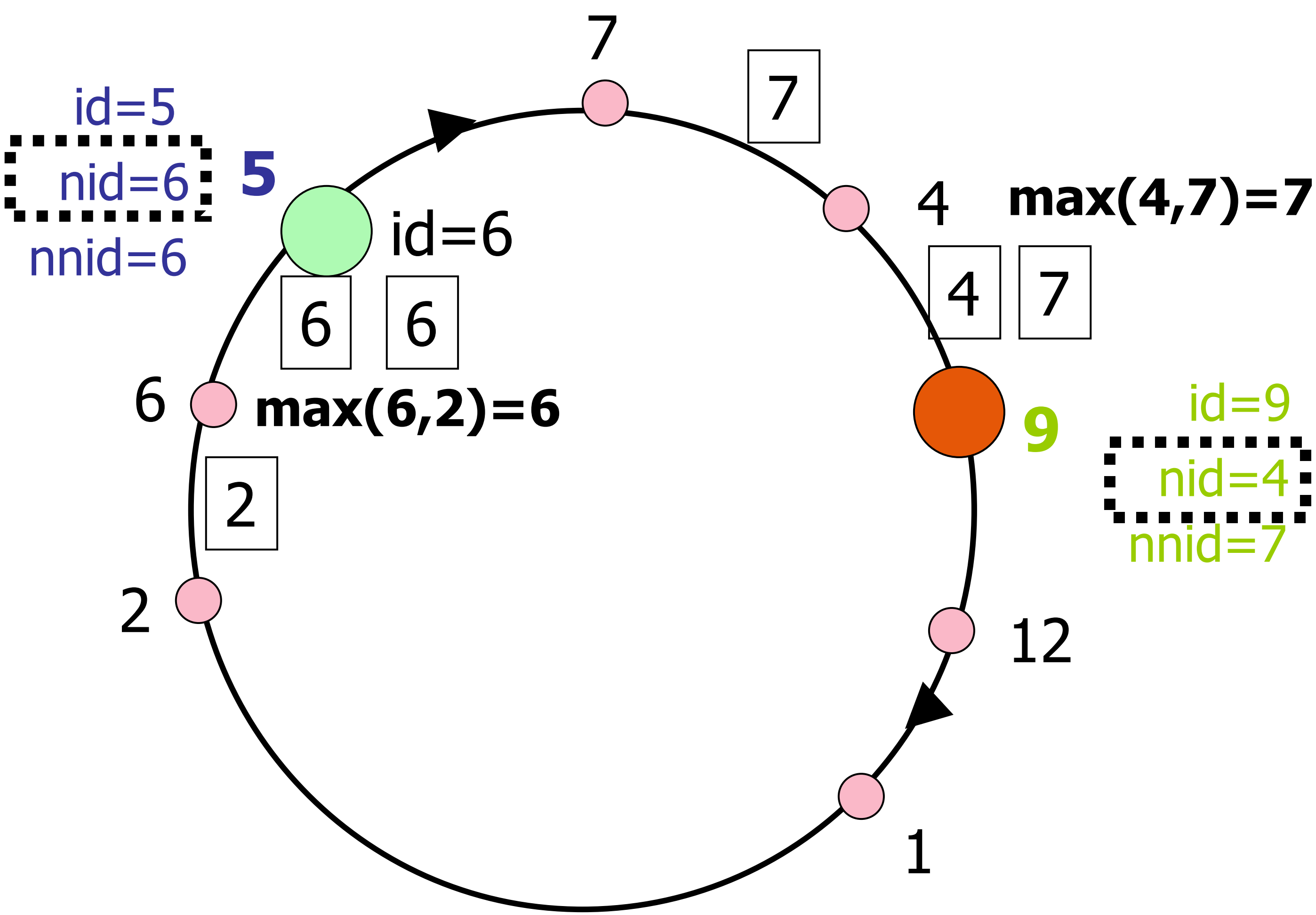
Peterson’s election algorithm (1/8)

- Is a **<u>simulation in a unidirectional ring</u>** of solution 2 in a bidirectional ring
- Every process receives the ids of **its two neighbors** in the "upstream" direction, and then **acts as its upstream neighbor**
- In round 1, every process
  - sends its **id** to its (downstream) neighbor
  - receives in variable **nid** the id of its (upstream) neighbor
  - sends **max(id,nid)** to its neighbor
  - receives this value in variable **nnid**
  - if **nid≥id** and **nid≥nnid**, it remains active and sets **id=nid**
  - otherwise turns **passive** (only relays messages in subsequent rounds)



TU Delft

Peterson's election algorithm (2/8)

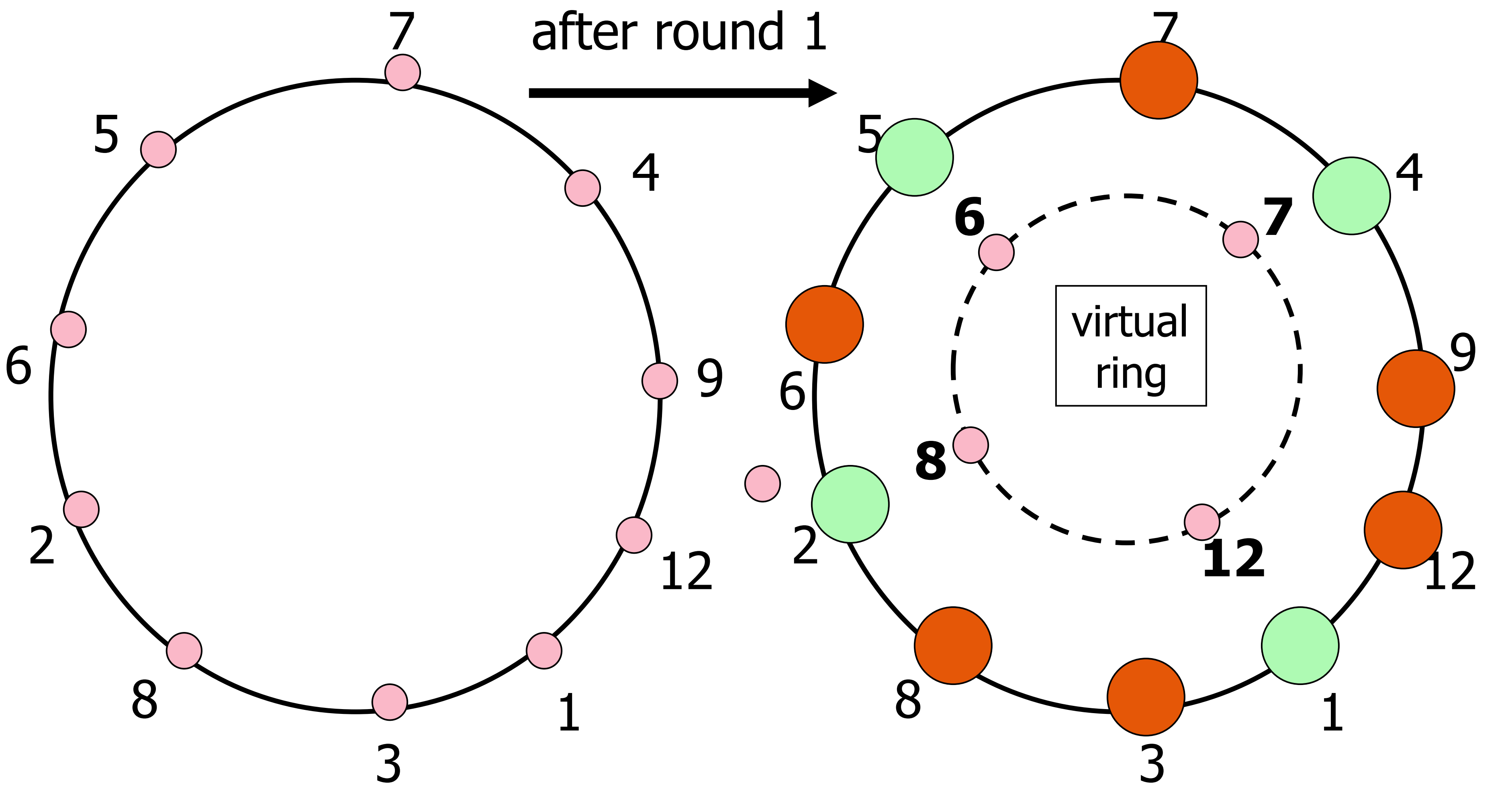


TU Delft

Peterson's election algorithm (3/8)

- After round 1, a **virtual ring** of active processes remains
- In **every subsequent round**, the algorithm of round 1 is repeated in the virtual ring of active processes
- The process that receives its own id has been elected

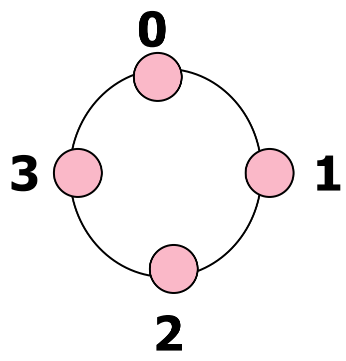
Peterson's election algorithm (4/8)



Peterson's election algorithm (5/8)

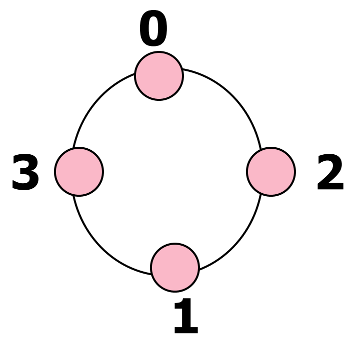
- **Question 1**: what is the message complexity?
- **Answer**: **2n·log(n)**

- **Question 2**: for which arrangement of ids along the ring does the algorithm terminate after one round?



- **Answer**: increasing or decreasing order

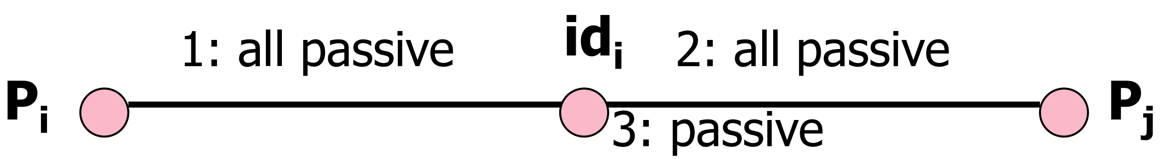
- **Question 3**: for which arrangement of ids along a ring of size $\mathbf{n=2^k}$ does the algorithm use **k** rounds?



TU Delft

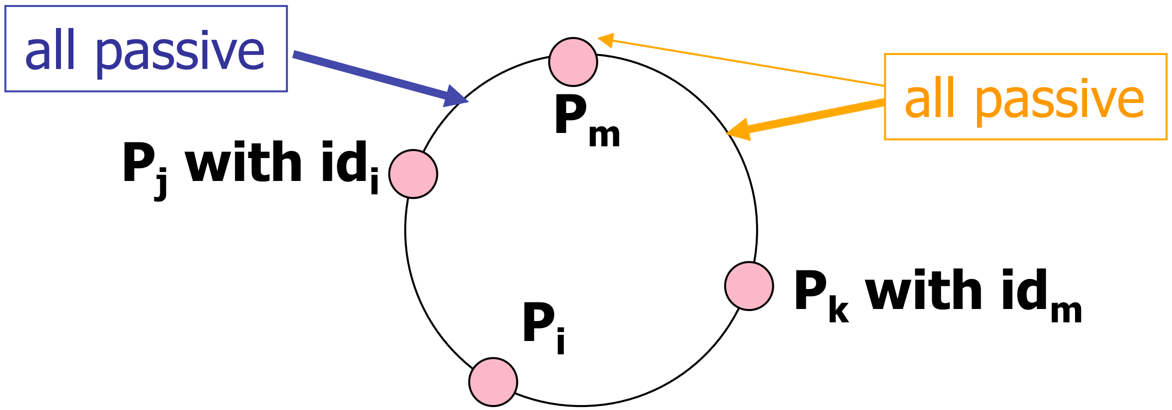
Peterson's election algorithm (6/8)

- **Correctness**: clearly, the maximum id survives
- **Only other issue**: no other process concludes that is has been elected
- **Assertion**:
  - if the id of process $\mathbf{P_i}$ **($\mathbf{id_i}$)** still survives in some active process $\mathbf{P_j}$ then **all processes between $\mathbf{P_i}$ and $\mathbf{P_j}$ are passive** (including $\mathbf{P_i}$)
- **Proof**:
  1. use induction
  2. ids are relayed by passive processes until the next active process
  3. a process whose id survives some round, becomes passive



Peterson's election algorithm (7/8)

- Let $\mathbf{P_m}$ be the process with maximal id ($\mathbf{id_m}$)
- **Suppose**:
  - $\mathbf{id_i}$ survives,
  - and gets to the last active process, say $\mathbf{P_j}$, before $\mathbf{P_m}$
  - $\mathbf{id_m}$ is in $\mathbf{P_k}$
- Then all processes between $\mathbf{P_j}$ and $\mathbf{P_k}$ are passive
- So $\mathbf{P_j}$ and $\mathbf{P_k}$ are neighbors, and $\mathbf{id_i}$ will not survive the next round



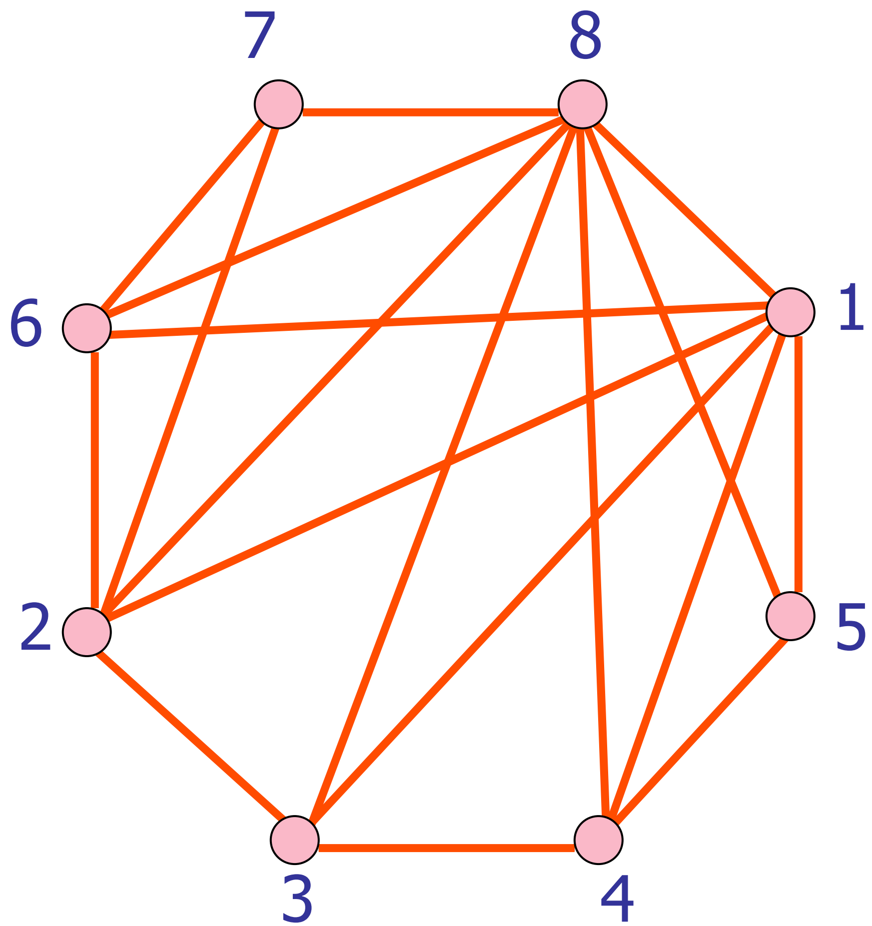
TU Delft

Peterson's election algorithm (8/8)

- **Time complexity** of an asynchronous distributed algorithm:

  the **length of the longest chain of messages** in any execution of the algorithm
- Time complexity of Peterson's algorithm with **n** processes: **2n-1**

Election in a complete network



Afek's and Gafni's synchronous algorithm (1/11)

- Assume a **synchronous system** and a **complete network**
- **Straightforward solution**: every process sends its id directly to everybody else: **n^2** messages and constant time
- **Main idea of A&G**:
  - **cut back** on the number of messages by successively sending an id to **ever larger sets** of processes and waiting for an ack from all of them
- A&G's algorithm is **message-optimal: $2n \cdot \log(n)$** messages
- Number of rounds: **$2 \cdot \log(n)$**
- A&G's algorithm is as fast as a message-optimal algorithm can be

TU Delft

Afek's and Gafni's synchronous algorithm (2/11)

- **Outline:**

1. in successive rounds, **alive candidate processes** send messages to **ever larger subsets of processes**

2. when a process receives an id larger than its own, it sends an **acknowledgments** back

3. meaning of **ack** = "you are bigger" (the process sending the ack is **captured** and is **owned** by the sending process)

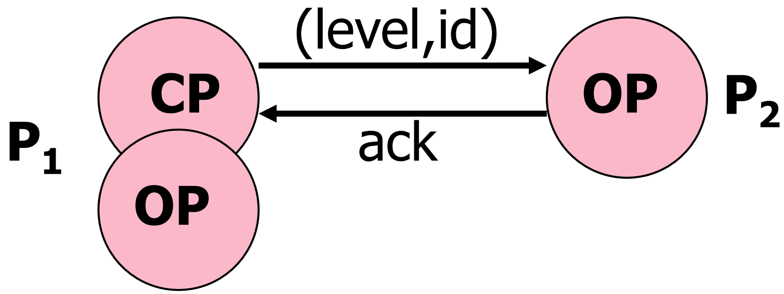
4. a candidate process that **does not receive all acks** it **expects in a certain round**, is **killed** **Synchronous algorithm!!!**

5. a process **adopts the largest id** it has ever seen (which is the id of its owner)

TU Delft

Afek's and Gafni's synchronous algorithm (3/11)

- Multiple nodes may start the algorithm **spontaneously** in different rounds
- A node that does so, spawns
  - a **candidate process**
  - an **ordinary process**



- A node that awakens due to the reception of a message only spawns an **ordinary process**
- Both types of processes keep track of their **level**, which is the **number of rounds since their start**
- Candidate processes send messages of the form **(level,id)** to ordinary processes in other nodes
- Ordinary processes send **acks** to candidate processes

Afek's and Gafni's synchronous algorithm (4/11)

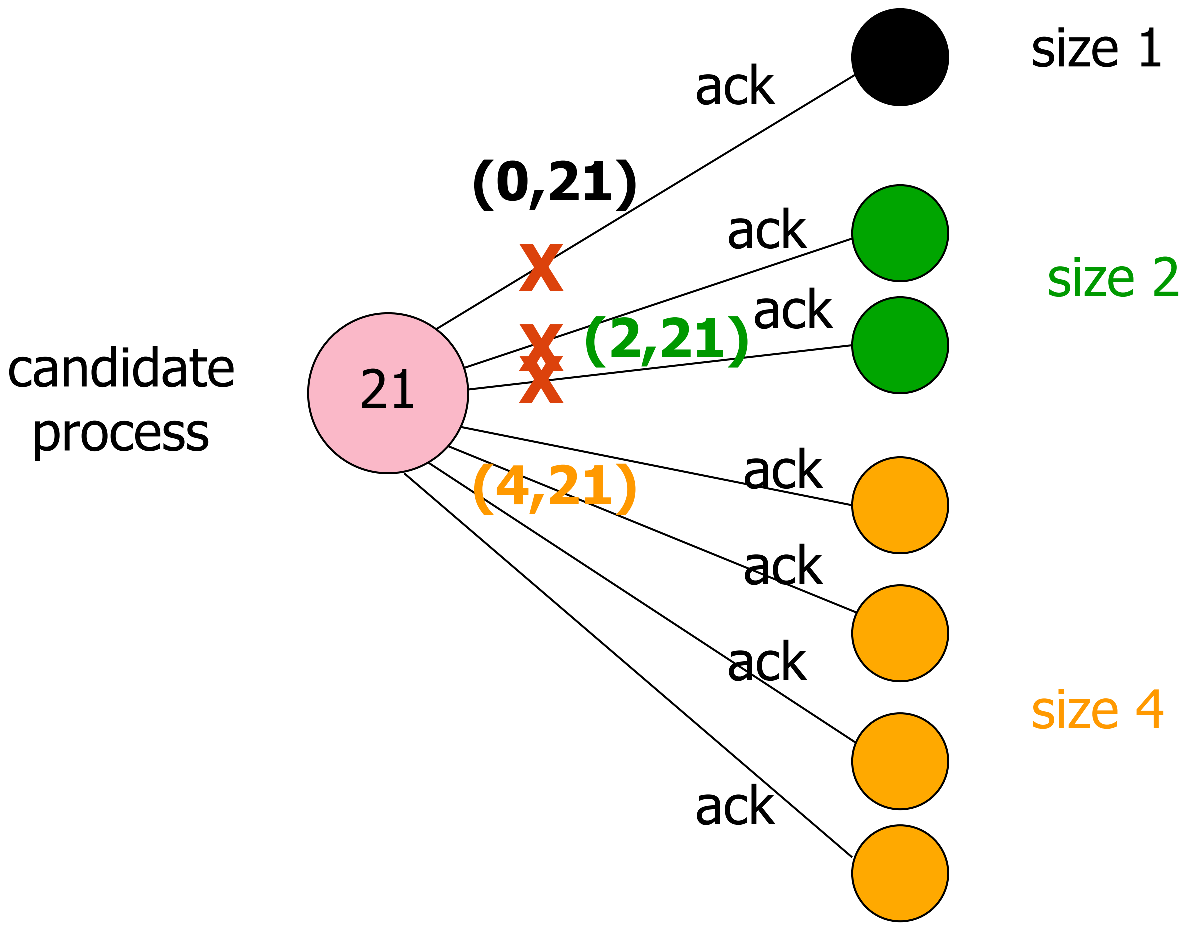
- Candidate processes send **candidate messages** with their id to **ever larger subsets** of sizes powers of 2, **as long as they receive all acks**
- Nodes keep track of all their **remaining links:**
  - no id sent or received over it
  - only these may be included in a future subset
- In fact, the node with the largest id **among those that started the election earliest wins**
- So **lexicographically largest** among all pairs (level,id):

$$\boxed{(\mathbf{2},5) > (\mathbf{1},3001)}$$

- **Question**: how does a node knows it has been elected?

TU Delft

Afek's and Gafni's synchronous algorithm (5/11)



Afek's and Gafni's synchronous algorithm (6/11)

- **Candidate process**:

```
E := set of all links of the node
level := -1
do forever
    level := level + 1                          /* round counter */
    if (level is even) then
        if (E = ф) then
            ELECTED                             /* all links have been used */
        else
            K := min(2^(level/2),|E|)           /* subsets of increasing size */
            E' := any subset of E of K elements
            send(level,id) over all links in E'
            E := E \ E'                         /* delete links used */
    else
        A := set of all acks received
        if (|A|<K) then STOP                    /* not all acks received */
```

send id

receive acks

TU Delft

Afek's and Gafni's synchronous algorithm (7/11)

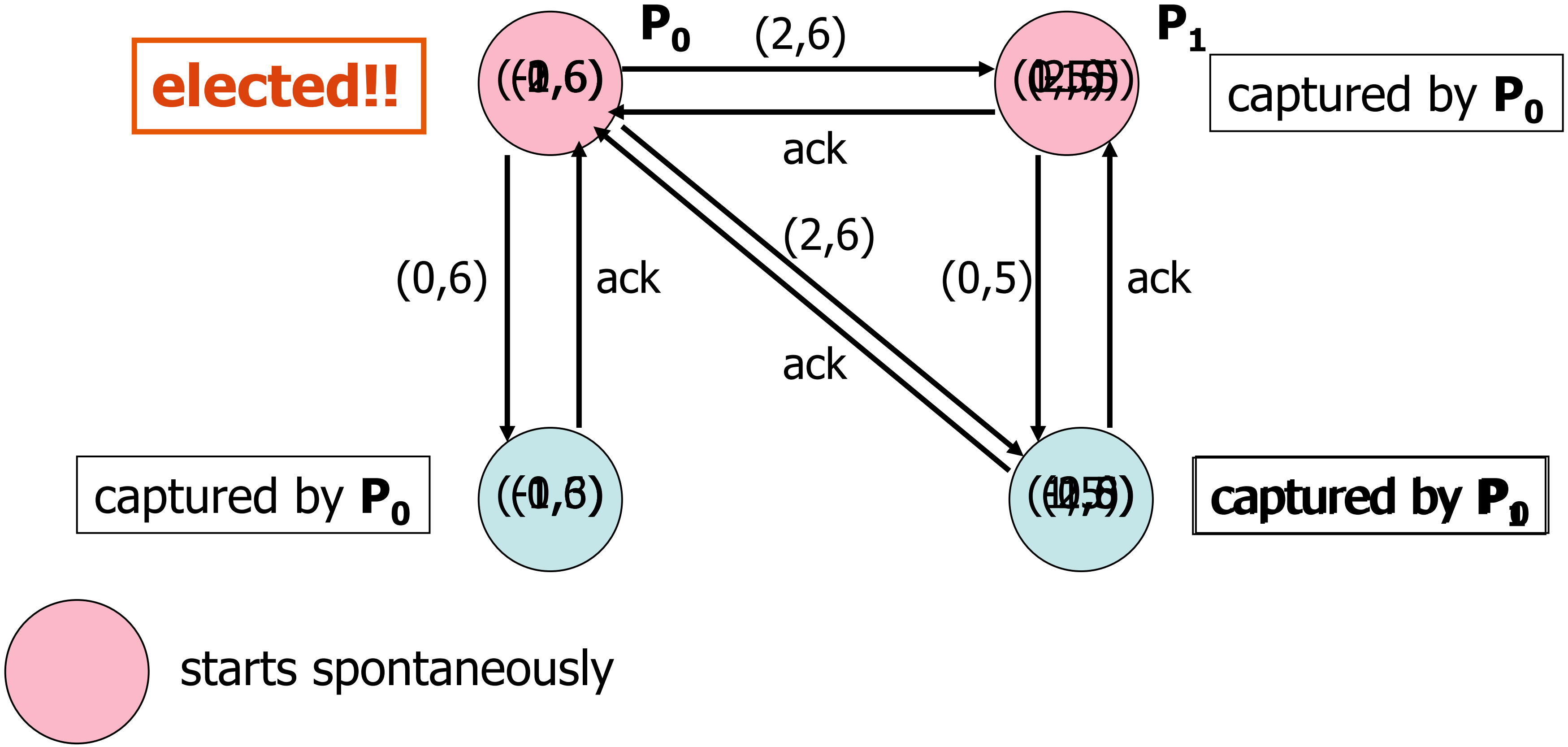
- **Ordinary process**:

```
link := nil
level := -1
do forever
    send(ack) over link                  /* at most one ack sent */
    level := level + 1
    R := set of all candidate messages received
    (nlevel,nid) := lexicographic maximum in R
    if ( (nlevel,nid) > (level,id) ) then /* other one is bigger */
        (level,id) := (nlevel,nid)       /* adopt biggest one seen */
        link := link over which (nlevel,nid) was received
    else                                 /* so only send ack along link */
        link := nil                      /* with highest known (l,i) */
```


TU Delft

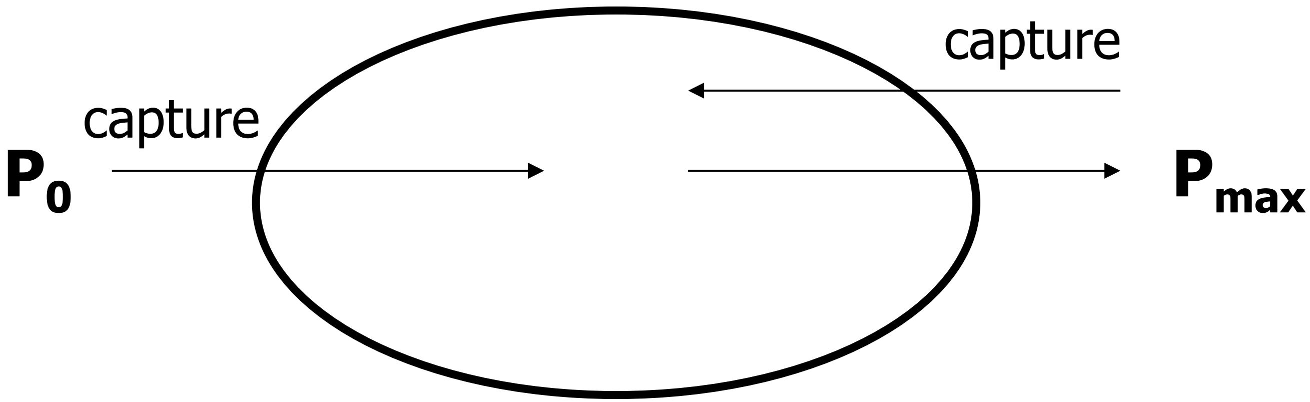
Afek's and Gafni's synchronous algorithm (8/11)

- **Example.** Notation: (level, id)



Afek’s and Gafni’s synchronous algorithm (9/11)

- Using levels is not strictly needed
- **Question**: What is the disadvantage of not using levels?
- **Answer**: the time needed for the algorithm may increase because the future winner starts later or first has to be woken up



TU Delft

Afek's and Gafni's synchronous algorithm (10/11)

- **Time complexity**:
  - the eventual winner will successively capture 1, 2, 4, 8, 16, … processes
  - so this stops after **log(n)** (double) rounds

TU Delft

Afek's and Gafni's synchronous algorithm (11/11)

Message complexity:

- every process sends at most one ack in every (double) round, so at most **n·log(n) acknowledgments**
- in (double) round **i**, the set of processes captured by a surviving candidate process has size $\mathbf{2^{i-1}}$
- the **sets of processes** captured by the surviving candidates in the same round are **disjoint**
- so there are at most $\mathbf{n/2^{i-1}}$ candidate processes left after (double) round **i**
- so the **total number of candidate messages** is at most

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

- so in total **2n·log(n)** messages

TU Delft

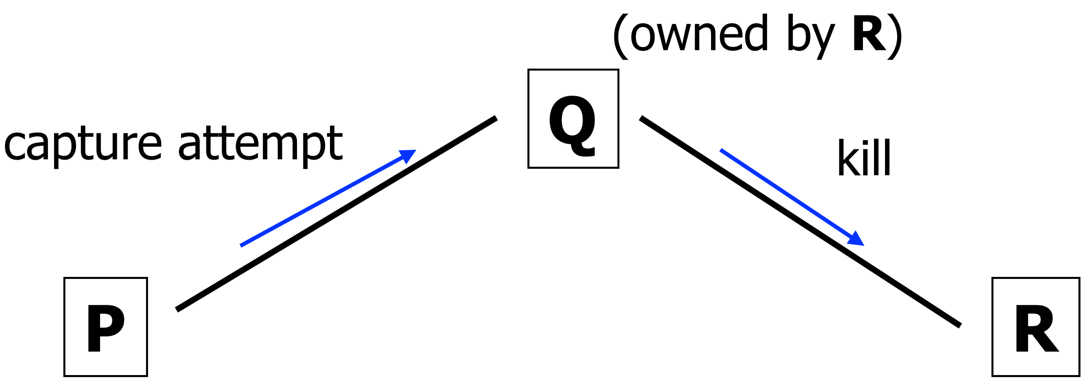
Afek' s and Gafni's asynchr. algorithm (1/6)

- Assume now an **asynchronous system** with a **complete network**
- It does not make sense for a node to wait for "all" messages it will receive to select the largest id
- So it might just as well react as soon as it receives a **single message**
- So let nodes try **to capture** other nodes **one at a time**
- The **level** of a node is now used to indicate **the number of nodes it has captured**

TU Delft

Afek's and Gafni's asynchr. algorithm (2/6)

- When a node **P** captures a node **Q** that is currently owned by node **R**, node **R** has to be killed because:
  - it will not win anyway
  - it will otherwise continue to try capturing, leading to wasted messages
  - its level will be wrong (**Q** is taken away from it)

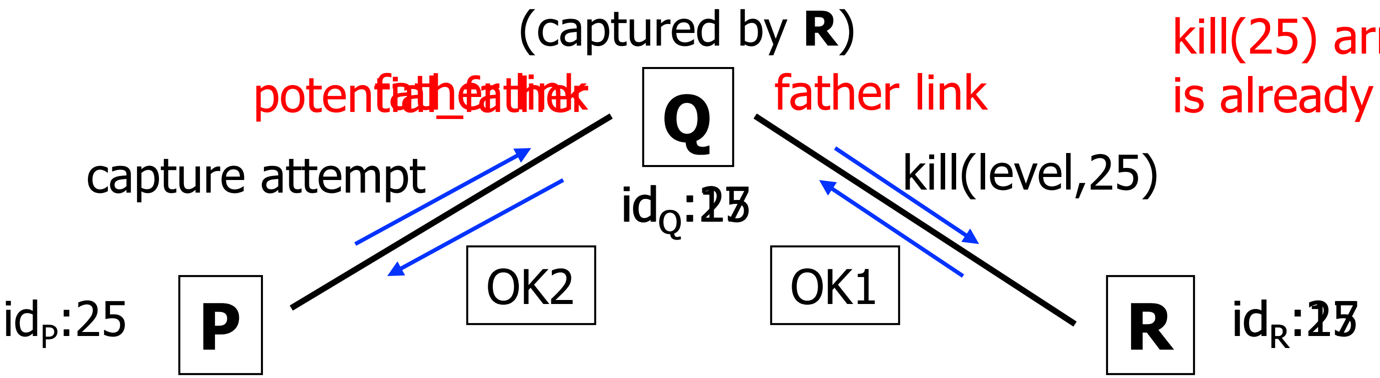


TU Delft

Afek's and Gafni's asynchr. algorithm (3/6)

- A node keeps two special links:
    1. **father** (link to owner)
    2. **potential_father** (link to potential new owner)
- A node about to be captured **will try to kill its father** on behalf of the new capturing node
- When a node (**R**) is attempted to be killed, **it may not be a candidate anymore** (already killed by another node)
- A node that has captured many nodes may be "killed" **many times**

- OK1 is not sent if before kill(25) arrives at **R**, id_R is already larger than 25



Afek's and Gafni's asynchr. algorithm (4/6)

- **Candidate process**:

```
while (untraversed ≠ φ) do
      link := any untraversed link
      send(level,id) on link                          /* attempt to capture */
R:    receive(level',id') on link'                    /* may be a kill attempt */
      if ( (id'=id) and (not killed) ) then           /* own value: ack */
                level := level+1                      /* capture succeeded */
                untraversed := untraversed – link
      else      if ( (level',id') < (level,id) ) then goto R  /* discard message */
                else
                          send(level',id') on link'   /* send ack back */
                          killed := true
                          goto R
enddo
if (not killed) then ELECTED
```

Initializations:
level = owner_id=0
untraversed = all links
father=nil
killed=false

TU Delft

Afek's and Gafni's asynchr. algorithm (5/6)

- **Ordinary process:**

```
do forever
        receive(level',id') on link'
        case (level',id') of
                • (level',id')<(level,owner_id):        /* simply ignore */
                • (level',id')>(level,owner_id):
                        potential_father := link'        /* potential new owner */
                        (level,owner_id) := (level',id')
                        if (father=nil) then father := potential_father
                        send(level',id') on father link /* kill or ack */
                • (level',id')=(level,owner_id):        /* ack from previous father */
                        father := potential_father      /* change father */
                        send(level',id') on father link /* ack new father */
enddo
```


TU Delft

Afek's and Gafni's asynchr. algorithm (6/6)

- Time complexity is **O(n):**
  - candidates don't wait for each other
  - 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
  - the winner captures nodes one by one
- Message complexity is **n·log(n)**

TU Delft