UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

Distributed Artificial Intelligence

CdLM Ingegneria Informatica

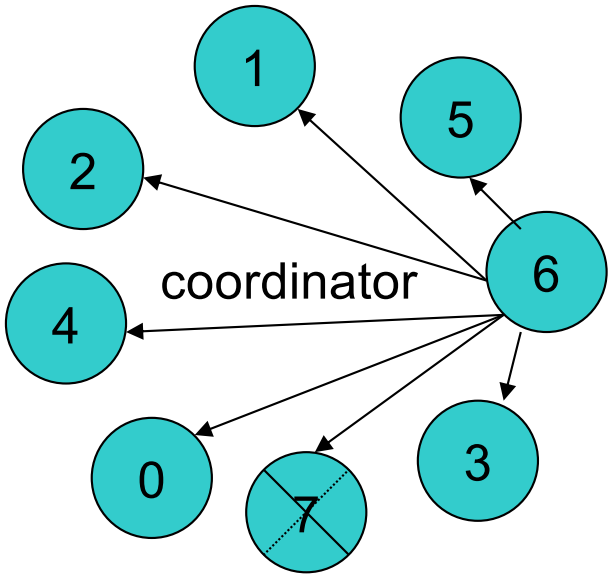
Prof. Franco Zambonelli

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Distributed Algorithms: Leader Election and Distributed Aggregation

Franco Zambonelli

October 2023



Outline

- Election algorithms – introduction
- Traditional election algorithms
- Dynamic (wireless) election algorithms
- From leader election to distributed aggregation
- Distributed aggregation algorithms

UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

Introduction to Leader Election

Need for a Leader

- Many problems used in DAI and distributed systems require identifying a "coordinator" or "leader"
  - Who coordinate access to a resource?
  - Who executes a specific task Tx?
  - Who act as observer for global predicate evaluation?
  - Or when a group of processes/agents needs to be represented as a single identity
- Two situations
  - All processes/agents in the DAI system are equally suitable for the role
  - Some processes/agents may be in a situation to act better than other for the role

Election Algorithms

- Election algorithms are executed to identify a leader
- Any process can “call an election” (i.e., initiate the algorithm to choose a new coordinator).
  - There should no harm (other than extra message traffic) in having multiple concurrent elections.
- Elections may be needed when
  - The system is initialized and there is not design-time designated leader
  - If current coordinator crashes or retires
  - Whenever there is the need to execute a specific task and there is need of deciding who will execute it.
- Different elections might by needed to identify multiple leader for playing different roles
  - E.g., one leader to act as controller of a mutually exclusive resource and another leader to acts as monitor of consistent global states

Requirements

- When the election algorithm terminates a single process must have been selected as leader and every process must know and agree on its identity.
  - Shared decision
  - Shared perception
- Formalize: every process p_i in the system has a variable e_i to hold the ID of the elected coordinator
  - $\forall i$, e_i = undefined or e_i = L, where L is the non-crashed process elected as coordinator
  - All processes (that have not crashed) eventually set e_i = L.
- **Concurrent elections** initiated by different processes (and referring to the same leader role) should lead to the same results

Assumptions of Traditional Leader Election Algorithms (1)

- Every process/site has a unique ID; e.g.
  - the network address + the port number
  - a process number
- Every process in the system should know which other processes are in the group
  - That is, they know the IDs of all other processes
  - Although they might not know not which processes are up or down, or which are currently available to act as leader.
  - Not very dynamic, not suitable for dynamic and large-scale systems: we should try to release this assumption
- The process with the highest ID number will be the new coordinator.

Assumptions of Traditional Leader Election Algorithms (2)

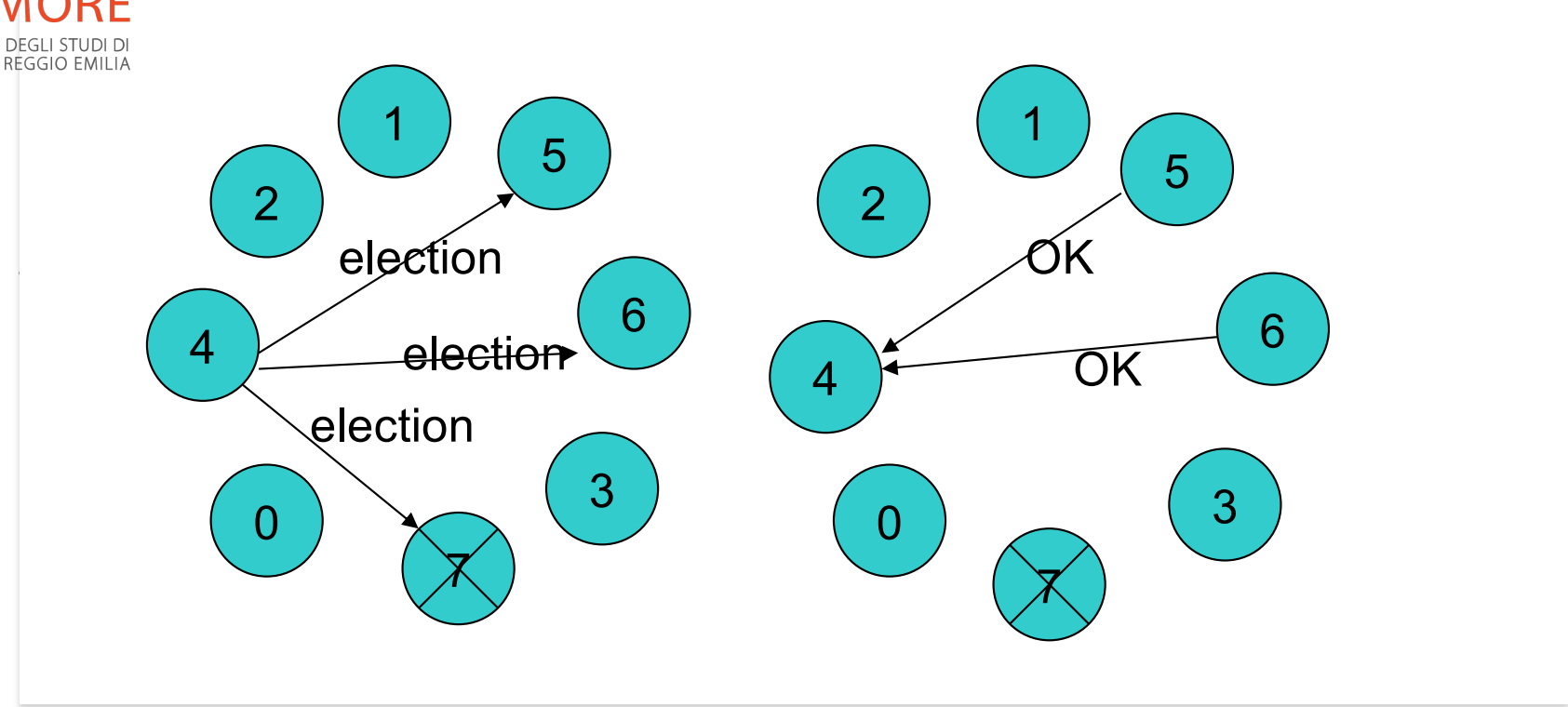
- Most algorithms assume that the process with the highest ID number will be the new coordinator
  - That is, the "property" that can make it the leader is its unique ID
- However, more in general, any local "unique property" of a process can be used in the following presented algorithms
- Examples
  - The process with the highest computational power
  - The process with the highest available bandwidth (if it has to act as coordinator of a shared resource)
  - The process/robot closest to a specific location
  - The process in the network that is a "hub", that is has the highest number of connections with other nodes
- More in general
  - Given function of the current state/environment of a process ***f(s, e)***
  - Find ***max(f(s_1,e_1), f(s_2,e_2),…,f(s_n,e_n))***

UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

Leader Election Algorithms

The Bully Algorithm - Overview

- Process *p* calls an election when it notices that the coordinator is no longer responding.
- High-numbered processes "bully" low-numbered processes out of the election, until only one process remains.
- When?
  - When a crashed process reboots, it holds an election. If it is now the highest-numbered live process, it will win.
  - When some process recognize that the current leader is no longer active, I calls an election

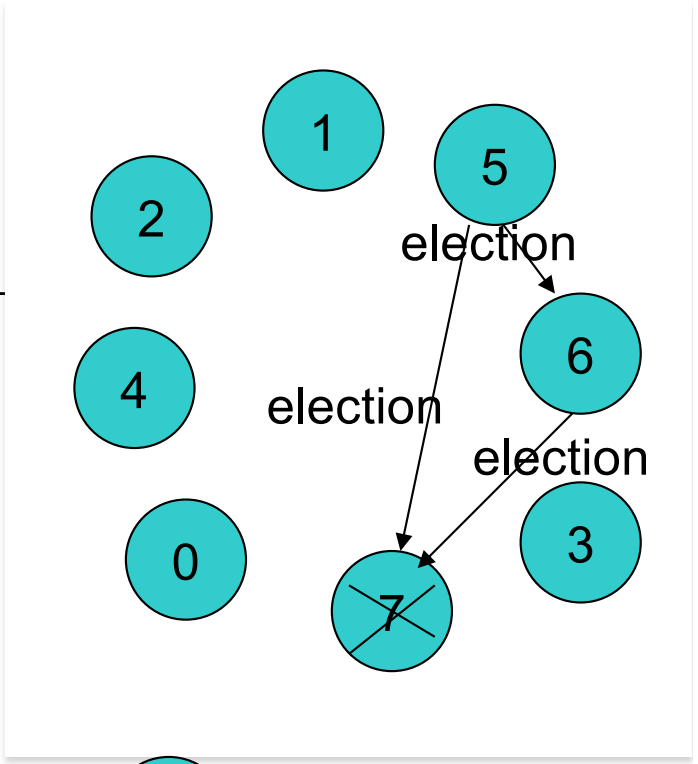


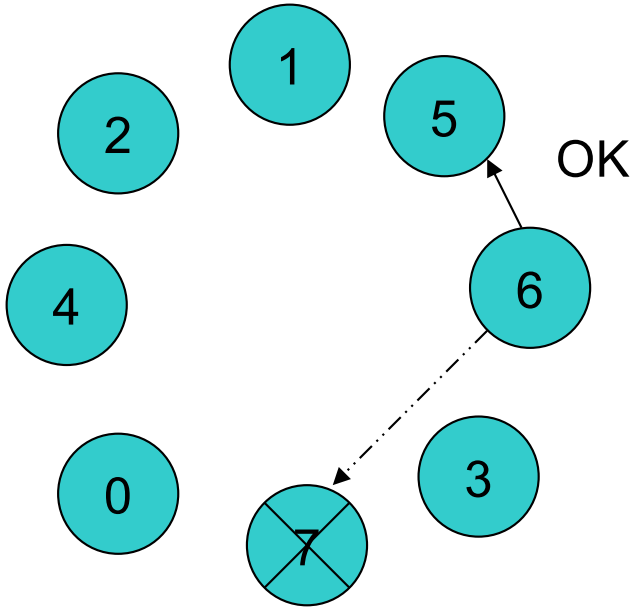
Process *p* sends an election message to all *higher-numbered* processes in the system.
If no process responds, then *p* becomes the coordinator.
If a higher-level process (*q*) responds, it sends *p* a OK message that terminates *p*'s role in the algorithm

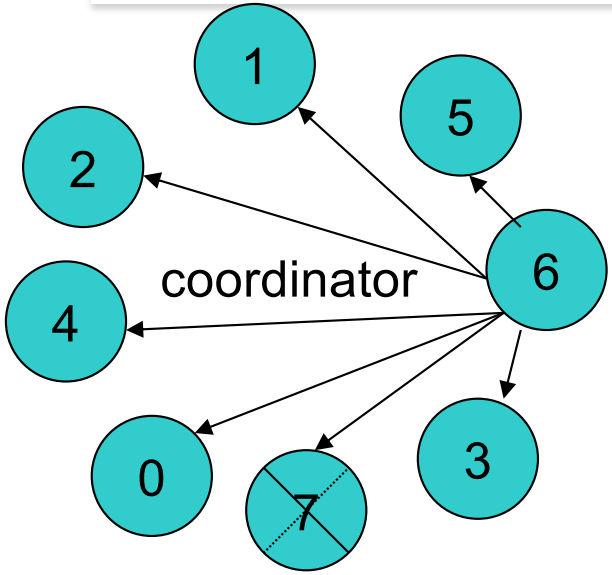
The process *q* now calls an election (if it has not already done so).

Repeat until no higher-level process responds. The last process to call an election "wins" the election.

The winner sends a message to other processes announcing itself as the new coordinator.







If 7 comes back on line, it will call an election

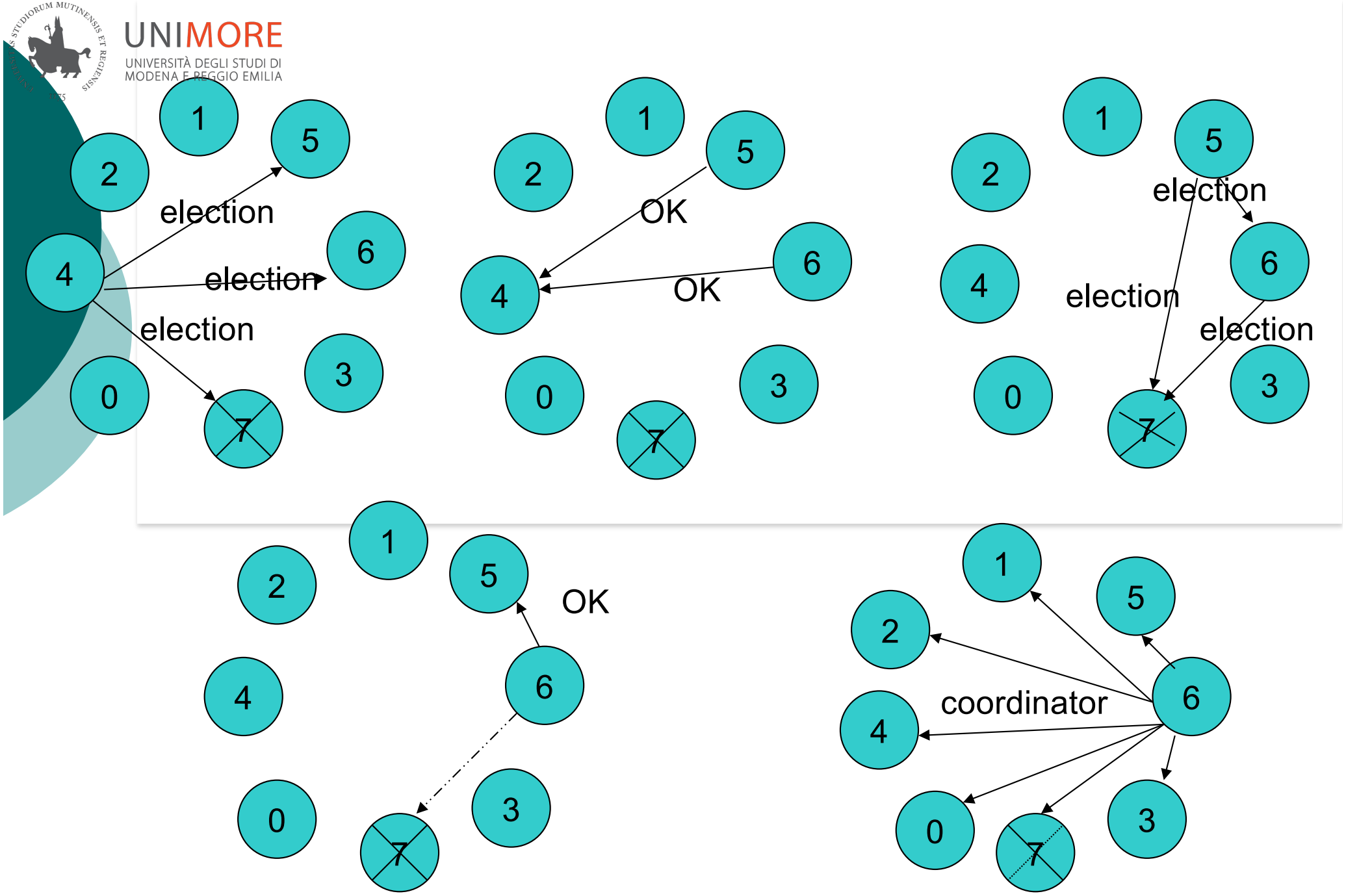

1
2
3
4
5
6
7
0
election
election
election
OK
OK
election
election
coordinator

Figure 6-20

Analysis

- Works best if communication in the system has bounded latency so processes can determine that a process has failed by knowing the upper bound (UB) on message transmission time (T) and message processing time (M).
  - UB = 2 * T + M
- However, if a process calls an election when the coordinator is still active, the coordinator will win the election.

Complexity

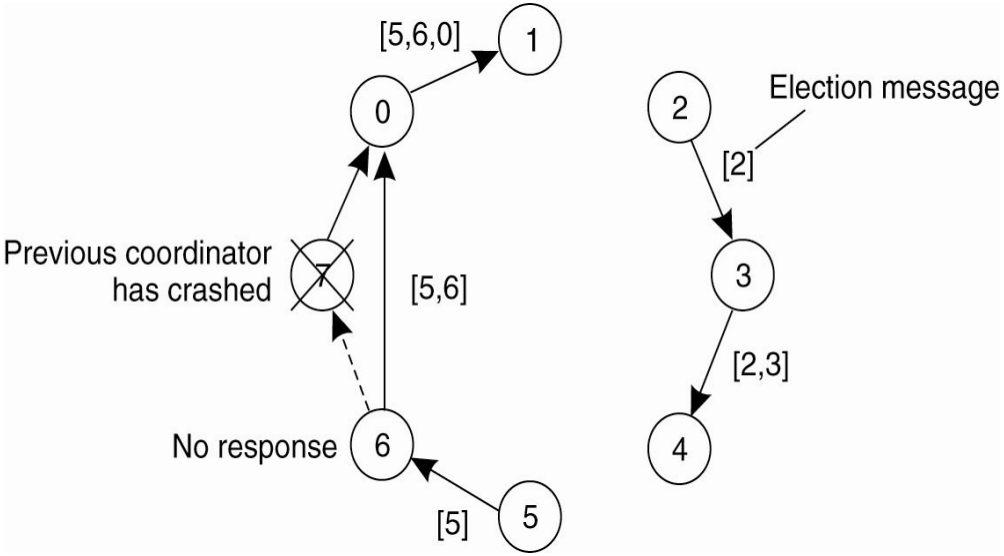
- The complexity (i.e., the number of messages required to complete) depends on who starts the election
- Given we have **n** processes
- If the election is started by the highest ID process
  - **n-1** messages are enough ("I will be the leader" said to everyone else!!)
- If the election is started by the lowest ID process
  - **n(n-1)** messages

The Ring Algorithm - Overview

- The ring algorithm assumes that the processes are arranged in a logical ring and each process is knows the order of the ring of processes.
- Processes are able to “skip” faulty systems: instead of sending to process `j`, send to `j + 1`.
- Faulty systems are those that don’t respond in a fixed amount of time.

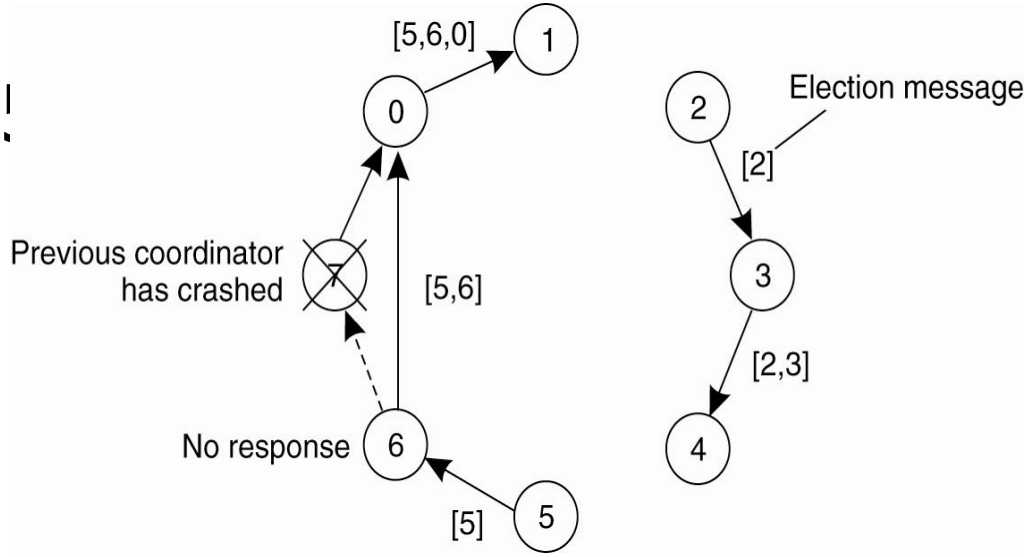
A Ring Algorithm

- P thinks the coordinator has crashed; builds an ELECTION message which contains its own ID number.
- Sends to first live successor
- Each process adds its own number and forwards to next.
- OK to have two elections at once.



Ring Algorithm - Details

- When the message returns to *p*, it sees its own process ID in the list and knows that the circuit is complete.
- P circulates a COORDINATOR message with the new high number.
- Here, both 2 and 5 elect 6:
  [5,6,0,1,2,3,4]
  [2,3,4,5,6,0,1]



Complexity

- Given we have **n** processes
- The complexity is always **2n**

Limitation of Traditional Leader Election Algorithms

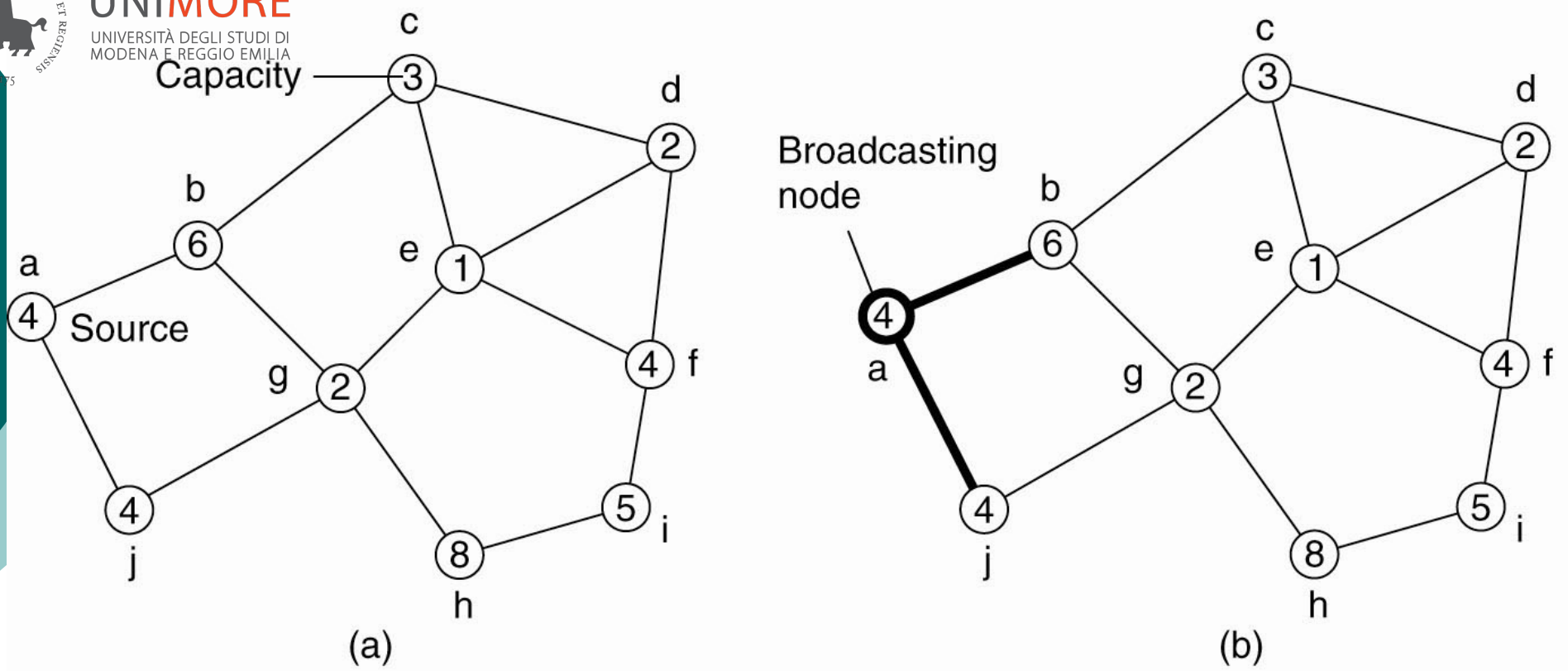
- Traditional algorithms aren't appropriate for wireless, dynamic, and large-scale systems.
  - They assume reliable message passing or stable network configuration, which is not guaranteed in wireless and dynamic systems
  - They assume nodes know each other a priori, which excludes scenarios in which processes come and go, and new processes can be created
  - They assume processes can communicate with all other processes, whereas in large systems typically a process communicate with a limited number of "neighbor" in the overall process social network
- We need novel network-oriented algorithms

Dynamic (Wireless) Leader Election Algorithm

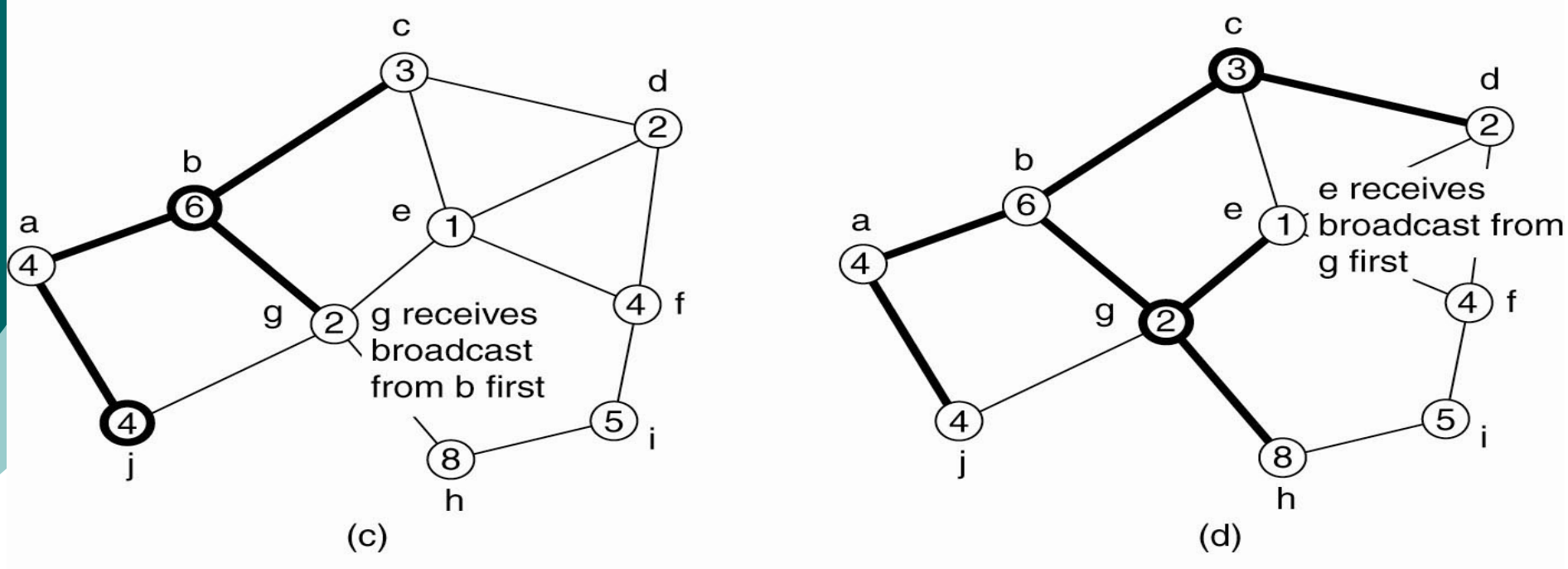
- The name of the algorithm focus on *ad hoc wireless networks* but, but in general such algorithm applies to
  - Wireless, unreliable networks, there included IoT (Internet of Things networks), with short range connections
  - Large-scale networks where processes interact with a limited number of neighbors only
  - Whenever nodes do not know a priori their IDs

Assumptions

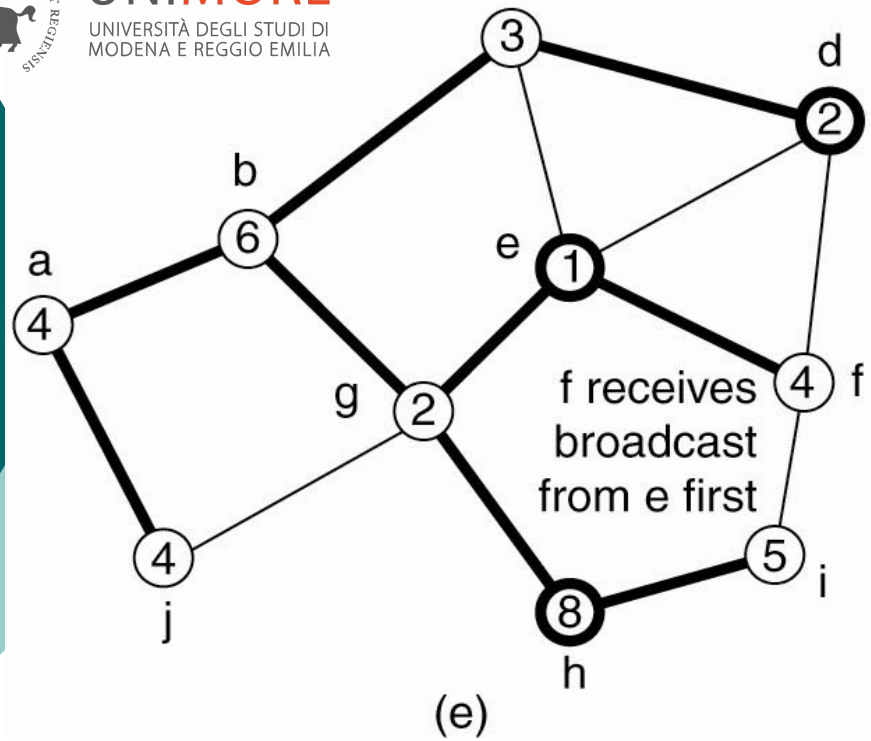
- For now, let us assume the node to become leader is still the one with the highest ID
- Wireless algorithms try to find the node suitable without knowing a priori each other IDs
- Any node (the source) can initiate an election by sending an ELECTION message to its neighbors – nodes within range.
- When a node receives its first ELECTION message the sender becomes its *parent node.*

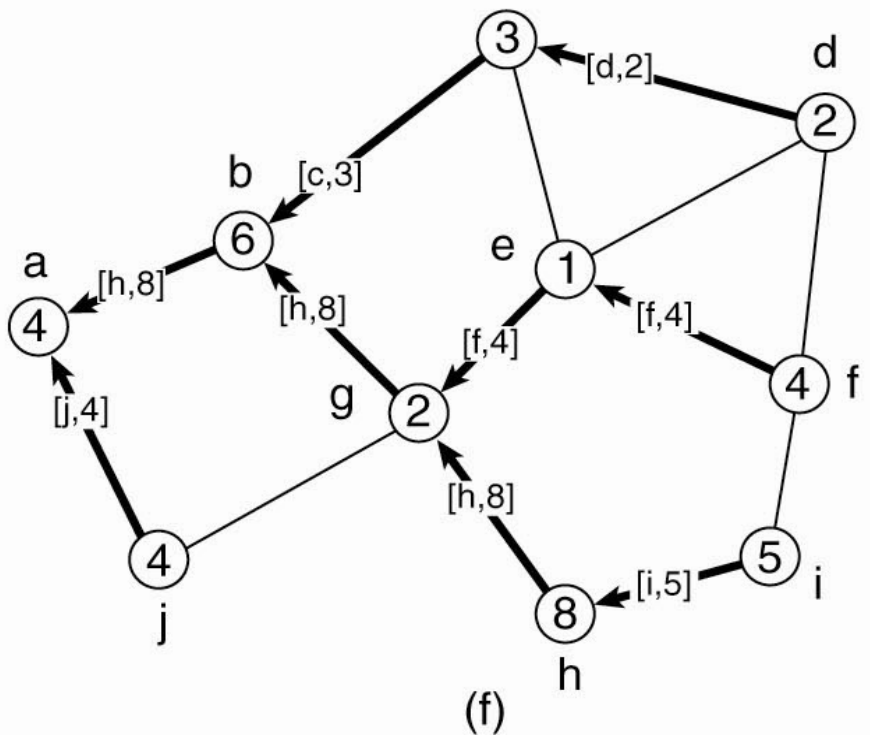


- Node **a** is the source.
  Messages have a unique ID to manage possible concurrent elections



When a node **R** receives its first election message, it designates the source **Q** as its parent, and forwards the message to all neighbors except **Q**. When **R** receives an election message from a non-parent, it just acknowledges the message → this process basically builds a so called “spanning tree” in the network





If **R**'s neighbors have parents, **R** is a leaf; otherwise it waits for its children to forward the message to their neighbors.

When **R** has collected acks from all its neighbors, it acknowledges the message from **Q**.

Acknowledgements flow back up the tree to the original source.

Wireless Elections

- At each stage the “most eligible” or “best” node will be passed along from child to parent.
- Once the source node has received all the replies, it is in a position to choose the new coordinator.
- When the selection is made, it is broadcast to all nodes in the network.

Wireless Elections

- If more than one election is called (multiple source nodes), a node should participate in only one.
- Election messages are tagged with a process **id**.
- If a node has chosen a parent but gets an election message from a higher numbered node, it drops out of the current election and adopts the high numbered node as its parent. This ensures only one election makes a choice.

Multiple Leaders in Different Roles

- When there has to be different leader each for playing a specific leading role
- For instance, let us suppose that we need both one leader for controlling a shared resource and another leader for monitoring the state of the computation
- Solution
  - We simply need to add an "election type" in the message exchanges during the execution of the algorithm
  - This is true for all presented algorithms

UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

Distributed Aggregation

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

From Leader Election to Distributed Aggregation

- As stated earlier
  - The "value" upon which to decide who is elected can be any variable/property/state hold by processes
- Formally:
  - Given a set of n processes P_i, each with a local state s_i
  - Find $max(s_1,....s_i,...s_n)$
  - And have this knowledge shared by all processes
- Or more in general,
  - Given the state ***s*** of processes and the value of some environmental variables ***e*** they observe
  - Find $\boldsymbol{max(f(s_1,e_1), f(s_2,e_2),...,f(s_n,e_n))}$
- In this case the leader election
  - Becomes de facto an algorithm for **distributed perception**, and in particular an algorithm for "aggregated distributed perception" or simply "distributed aggregation"
  - And can be generalized to finding properties other than "max"

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Principles of Distributed Aggregation

- Given a network of nodes/processes
  - There is an "aggregation" function *AF* to be calculated in a distributed way from the state s of the processes
  - $S_{aggr} = AF(s_1, \ldots, s_n)$
  - Or more in general, if we include also environmental variables observed by processes
  - $AF((f(s_1,e_1), f(s_2,e_2), \ldots, f(s_n,e_n))$ where ***f*** can be locally computed by each process
- Distributed Aggregation vs Global Predicate Evaluation
  - Distributed aggregation enables to evaluate "aggregated" perspectives on the system state
  - So it can be used to evaluate what is globally happening in the computation

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Key Issues of Distributed Aggregation

- What aggregation functions ***AF*** can be computed?
  - All leader election algorithms presented so far can compute ***MAX*** and ***MIN***
- Can we identify a more distributed and efficient algorithm that:
  - Is capable of computing more general aggregation function? (e.g., the average, or the standard deviation)
  - Requiring no ”initiators”, but in which all nodes participate continuously to the algorithm and all nodes can eventually perceive the value S_{aggr} of the aggregated function
  - With not necessity to build spanning trees? (the wireless leader election algorithm can indeed be used to compute more complex aggregation functions than MAX and MIN, but for large networks building and navigating spanning tree can be costly and time consuming)

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Key Idea: Gossiping!

(Jelasity Montresor and Babaoglu 2003)

- What if we conceive the process of aggregating distributed values as a process of gossiping?
- Two ”friends” processes
  - Processes that are “**neighbour**” in the social network of processes (they know their relative IDs or simply are in range of a wireless connection)
- Gossip each other about the aggregation function
  - Process Y: “Hey, process X, I have heard that the maximum is 42”
  - ”Process X: “No, dear friend Y, I have heard that the maximum is 54”
  - Both: “OK then we update/confirm our knowledge, and both agree that, as far as we know, the maximum is 54”
- And then, we the updated knowledge, they start gossiping with some other nodes….

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Basic Scheme of Distributed Aggregation

(Jelasity Montresor and Babaoglu 2003)

- Periodically:
  - Each process actively select a neighbour (even randomly)
  - Send the locally aggregated value of ***s*** to it (or more in general the aggregated value of ***f(s,e)***)
  - In tuen, receive the value as aggregated by its neighbour
  - Update the value by computing the aggregation function (aggregating its value with the value of the neighbor)
- And do the same upon passively receiving

do exactly once in each consecutive
δ time units at a randomly picked time
  $q \leftarrow$ GETNEIGHBOR()
  send s_p to q
  $s_q \leftarrow$ receive(q)
  $s_p \leftarrow$ UPDATE(s_p, s_q)

(a) active thread

do forever
  $s_q \leftarrow$ receive(*)
  send s_p to sender(s_q)
  $s_p \leftarrow$ UPDATE(s_p, s_q)

(b) passive thread

Distributed Aggregation of Max/Min

- The maximum value, is spread from process to process, eventually reaching all nodes in the network
- Similarly for the minimum

do exactly once in each consecutive
δ time units at a randomly picked time
  $q \leftarrow$ GETNEIGHBOR()
  send s_p to q
  $s_q \leftarrow$ receive(q)
  $s_p \leftarrow$ MAX (s_p, s_q)

(a) active thread

do forever
  $s_q \leftarrow$ receive(*)
  send s_p to sender(s_q)
  $s_p \leftarrow$ MAX (s_p, s_q)

(b) passive thread

Average Aggregation

- To compute the average value
  - Starting from the local value
  - And continuously computing the average with neigbhors
- It can be easily proved that each step contributes locally and globally diminishing the variance of the estimate of the overall average value
  - Convergence of the algorithm

do exactly once in each consecutive
δ time units at a randomly picked time
  $q \leftarrow$ GETNEIGHBOR()
  send s_p to q
  $s_q \leftarrow$ receive(q)
  $s_p \leftarrow (s_p + s_q)/2$

(a) active thread

do forever
  $s_q \leftarrow$ receive(*)
  send s_p to sender(s_q)
  $s_p \leftarrow (s_p + s_q)/2$

(b) passive thread

Other Aggregation Function

- What can be computed with this approach?
  - Max, Min, Average, Generalized Means, Variance, Sums, Products
- Counting nodes: how many nodes are there in the network?
  - Elect a leader
  - Have a leader with s=1 and all other nodes with s=0
  - Compute the average. If average 1/n this means there are n nodes in the systems
- Counting any property: How many nodes are there for which Predicate PR is true?
  - Compute the average of PR, if average 1/x there are x nodes for which PR holds

Handling Dynamic Values in Distributed Aggregation

- If the values *s* to be aggregated changes over time, the risk is that the computation will never converge
- Solution 1 (Jelasity and Babaoglu, 2005)
  - Divide time into "epochs", and recompute aggregation at each new epoch
  - Let all processes consider the "freezed" value of s at the beginning of the epoch
  - Compute aggregation with the freezed value
  - Problem: What if there is no notion of global time?
- Solution 2 (Bicocchi and Zambonelli, 2010)
  - Let the aggregate values slowly "evaporate" (lose value)
  - So that the aggregation algorithm continuously re-computes them with the up-to-date values

Distributed Aggregation and Network Structure

- What happens if
  - Some nodes are never selected in the neighbour selection process OR
  - Some nodes crash and do no longer participate in the aggregation algorithm?
- The aggregation algorithm works well is the computation of the aggregated values of process couples properly propagate in the whole network
  - That is, if the "gossip" propagates
- We have already discussed (see social networks lessons) that
  - Small world and scale free networks has a very low percolation threshold
  - This epidemies (and gossip) propagate very efficiently even if many nodes do not participate in the diffusion
  - Consequence: in these networks, distributed aggregation algorithms are very robust with respect to node failures
- In addition
  - Given that these networks have a short average distance between nodes (characteristics path length)
  - Distributed aggregation propagated and converges very quickly!!!!

Conclusions

- Leader election algorithms are the basis for many other algorithms
  - Also very important in decentralized and dynamic setting, where no a priori design time decision can be taken on «who does what?»
- Distributed aggregation algorithms are at the basis of «distributed perception»
  - To let all process share the same knowledge about what is happening
  - To compute efficiently a sort of «distributed information fusion»
  - To compute some limited form of Global Predicate Evaluation in a much simpler way that traditional approaches based on vector clocks or snapshots