Distributed Systems

Leader election & Failure detection

He Sun
School of Informatics
University of Edinburgh

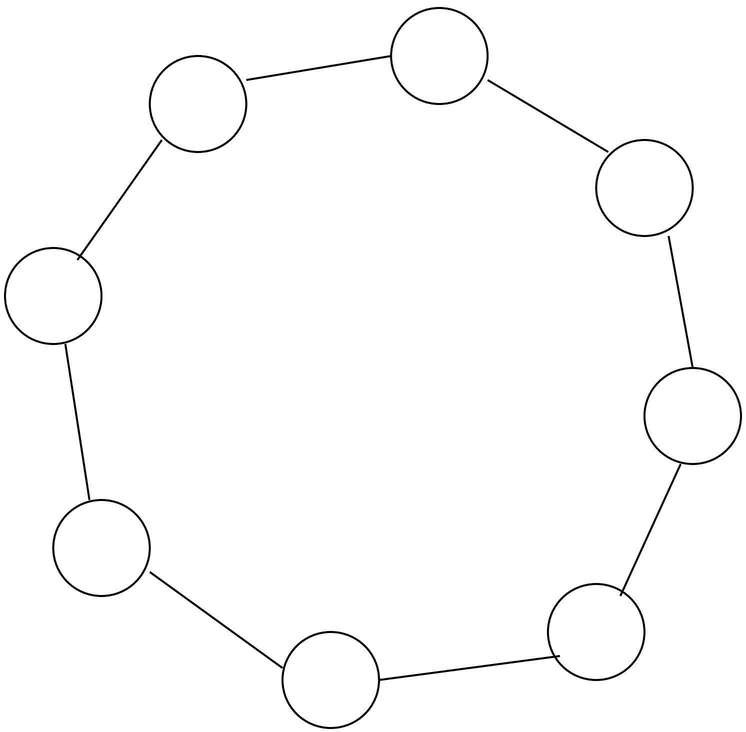
THE UNIVERSITY *of* EDINBURGH

Leader of a computation

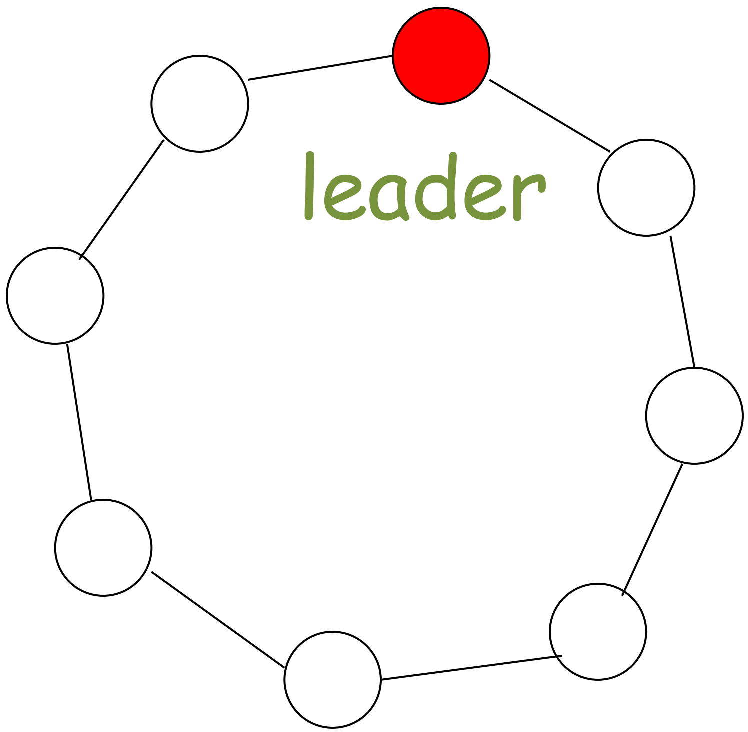
- Many distributed computations need a coordinator of server processors
  - E.g., Central server for mutual exclusion
  - Initiating a distributed computation
  - Computing the sum/max using aggregation tree
- We may need to elect a leader at the start of computation
- In every admissible execution, exactly ***one processor*** enters an elected state.
- We may need to elect a new leader if the current leader of the computation fails

Leader election in ring networks

Initial state (all not-elected)



Final state



Why study rings?

- Simple starting point, easy to analyze
- Lower bounds and impossibility results for ring topology also apply to arbitrary topologies

LE algorithms in rings depend on…

Anonymous Rings

Non-anonymous Rings

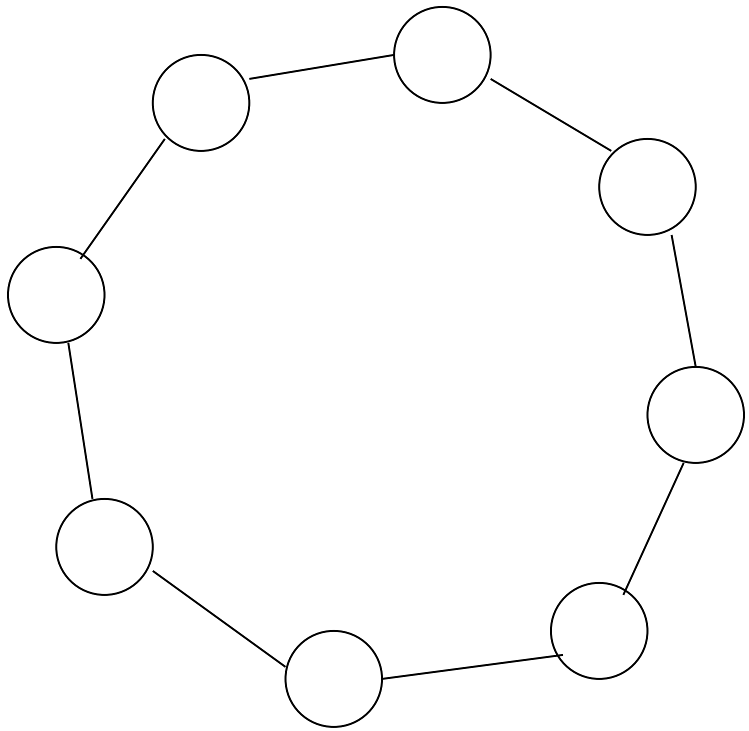
Size of the network n is known (non-unif.)

Size of the network n is not known (unif.)

Synchronous Algorithms

Asynchronous Algorithms

LE in Anonymous Rings



Every processor runs the same algorithm

Every processor does exactly the same execution

Impossibility for Anonymous Rings

Theorem

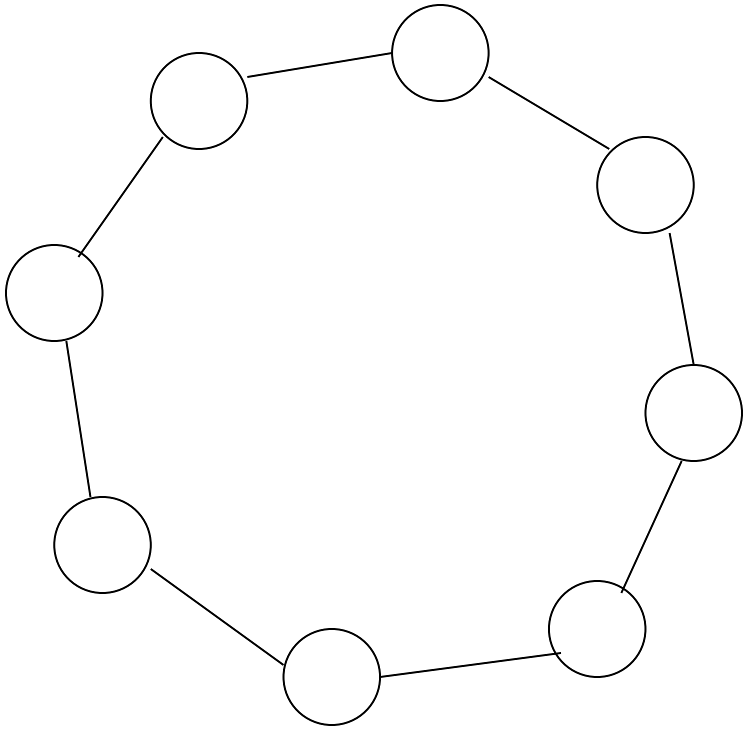
There is **no** leader election algorithm for anonymous rings, even if
- the algorithm knows the ring size (non-uniform)
- in the synchronous model

Proof Sketch (for non-unif and sync rings):

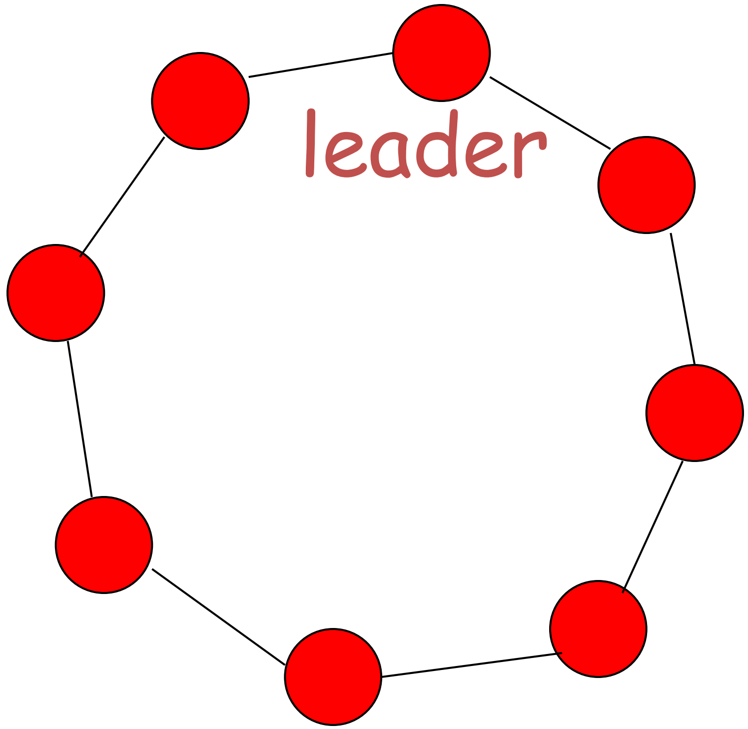
- Every processor begins in same state (**not-elected**) with same outgoing msgs (since anonymous)
- Every processor receives same msgs, does same state transition, and sends same msgs in round 1
- And so on and so forth for rounds 2, 3, …
- Eventually some processor is supposed to enter an elected state. But then they all would.

Initial state

(all not-elected)



Final state



If one node is elected a leader,
then every node is elected a leader

Impossibility for Anonymous Rings

Since the theorem was proven for non-uniform and synchronous rings,

the same result holds for ***weaker*** models:

- uniform
- asynchronous

Rings with Identifies (non-anonymous)

Assume each processor has a unique ID.

Do not confuse indices and IDs:

- indices are 0 to n-1: used only for analysis, not available to the processors
- IDs are arbitrary **nonnegative** integers: are available to the processors

Overview of LE in Rings with IDs

There exists algorithms when nodes have unique IDs. We will evaluate them according to their message complexity.

Best result:

- Asynchronous rings: $\Theta(n \log n)$ messages
- Synchronous rings: $\Theta(n)$ messages

All bounds are asymptotically tight!

Node with the highest identifier

- If all nodes know the highest ID (say n), we do not need an election.
    - Everyone assumes n is the leader
    - n starts operating as the leader
- But what if n fails? We cannot assume $n-1$ is leader, since $n-1$ may have failed too!

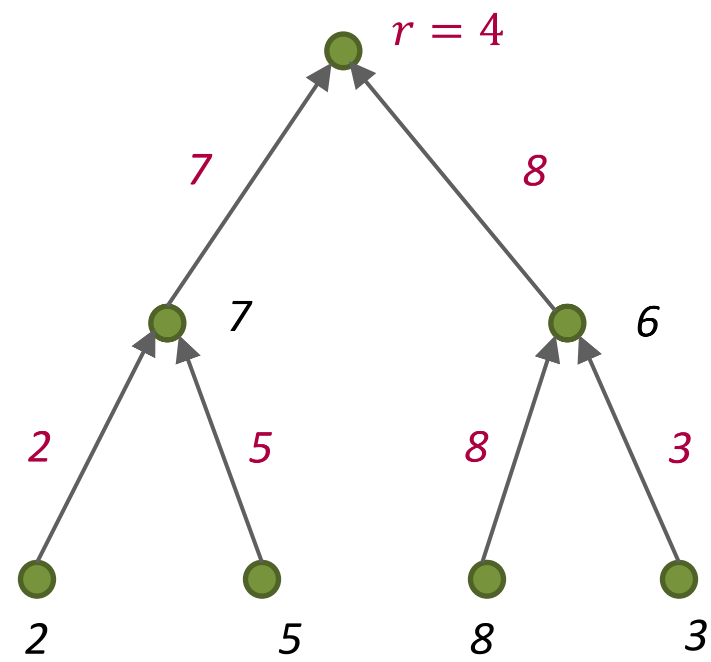
Our Strategy

The node with the highest ID and still surviving is the leader.

We need an algorithm that finds the working node with the highest ID.

One strategy: use aggregation tree

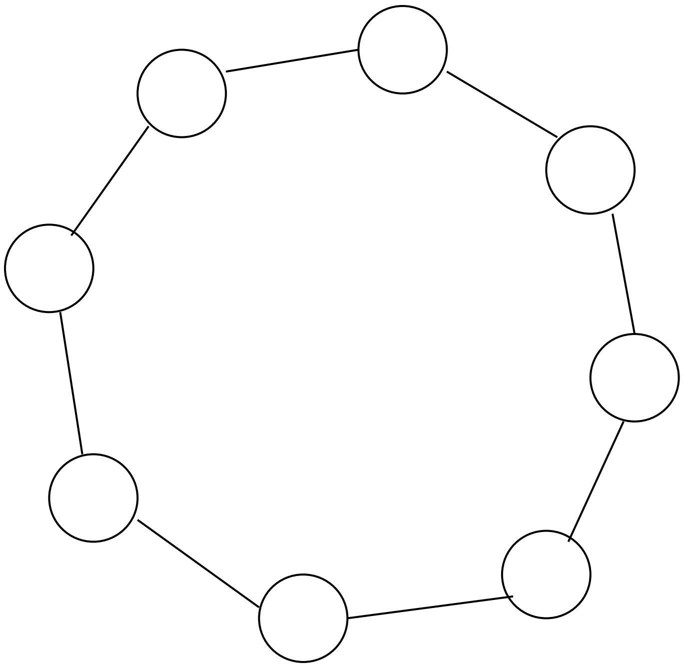
- Suppose node r detects that leader has failed, and initiates lead election
- Node r creates a BFS tree.
- Asks for max node ID to be computed via aggregation
    - Each node receives ID values from children
    - Each node computes *max* of own ID and received ID, and forwards to parents
- Needs a tree construction
- If n nodes start election, we'll need n trees
    - $O(n^2)$ communication
    - $O(n)$ storage per node



Can we do better?

Chang-Roberts algorithm: High-level Ideas

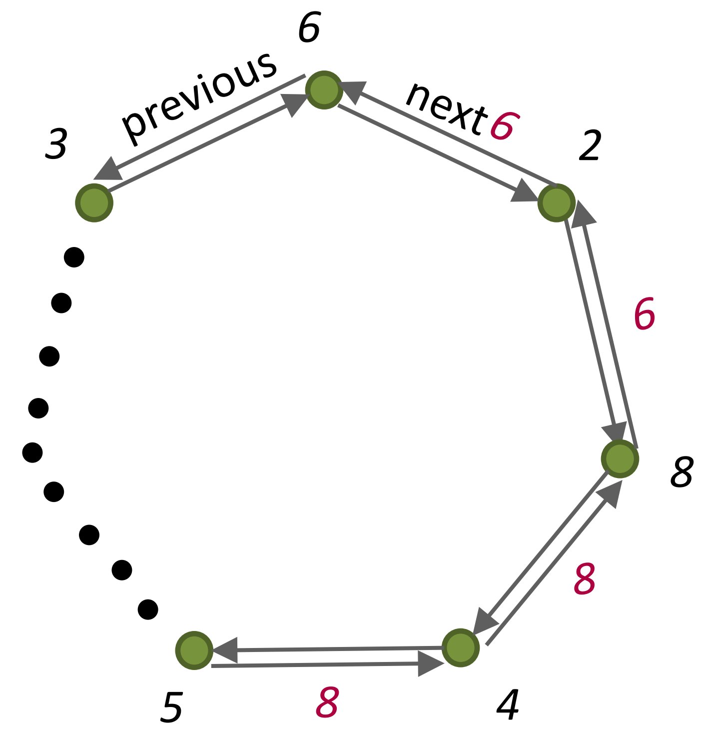
- Suppose the network is a ring
  - We assume that each node has 2 points to nodes it knows about
    - Next
    - Previous
    - (like a circular doubly linked list)
  - The actual network may not be a ring



- Every node send max(own ID, received ID) to the next node
- If a processor receives its own ID, it is the leader
- It is ***uniform***: number of processors does not need to be known to the algorithm

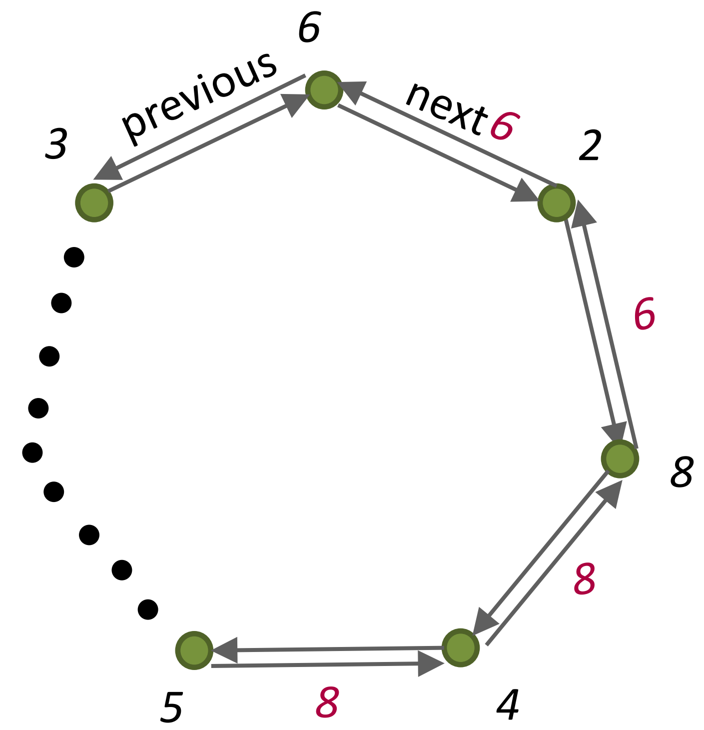
Chang-Roberts algorithm: example

- Basic idea:
  - Suppose 6 starts election
  - Send “6” to 6.next, i.e. 2
  - 2 takes max(2,6), sends to 2.next
  - 8 takes max(8,6), sends to 8.next
  - Etc



Chang-Roberts algorithm: example

- The value “8” goes around the ring and comes back to 8
- Then 8 knows that “8” is the highest ID
  - *Since if there was a higher ID, that would have stopped 8.*
- 8 declares itself the leader: sends a message around the ring.



Chang-Roberts algorithm : final step

- If node p receives election message m with m.ID=p.ID

- P declares itself leader
  - Set p.leader=p.ID
  - Send leader message with p.ID to p.NEXT
  - Any other node q receiving the leader message
    - Set q.leader=p.ID
    - Forwards leader message to q.NEXT

Chang-Roberts algorithm: discussion

- Works in an asynchronous system
- Correctness: Elects processor with largest ID
  - *msg containing that ID passes through every processor.*

- Message complexity $O(n^2)$
  - When does it occur?
  - Worst case to arrange the IDs is in the decreasing order:
    - 2nd largest ID causes $n - 1$ messages
    - 3rd largest ID causes $n - 2$ messages
    - Etc.
    - Total messages = $n + (n-1) + (n-2) + \ldots + 1 = O(n^2)$

Average case analysis of Chang-Roberts algorithm

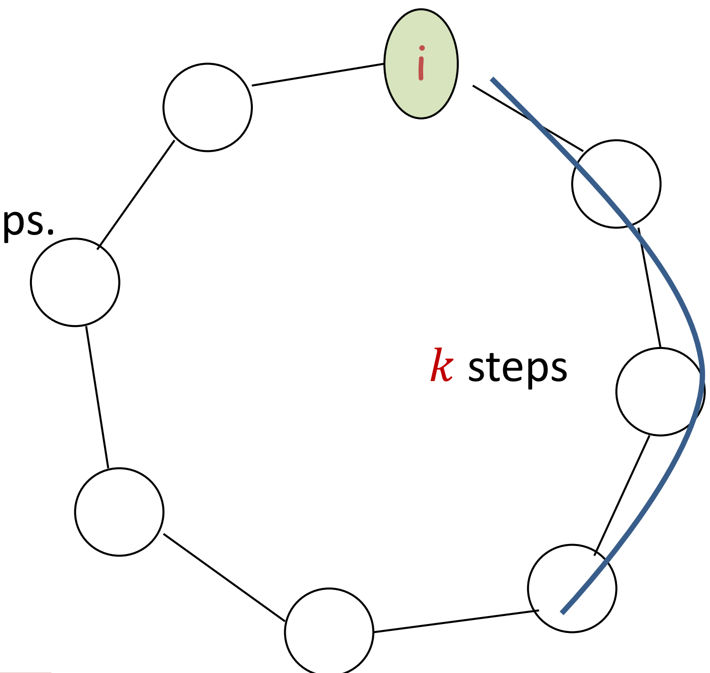
Theorem

The average message complexity of Chang-Roberts algorithm is $O(n\log n)$

Assume that all rings appear with equal probability.

For the proof, assume IDs are $1,2, \ldots\ n$

Let $P(i,k)$ be the prob. that ID $\boldsymbol{i}$ makes exactly k steps.



Prob. that the $k-1$ neighbors of i are < i

$$P(i,k) = \frac{\binom{i-1}{k-1}}{\binom{n-1}{k-1}} \cdot \frac{n-i}{n-k}$$

Prob. that the k neighbor of i is >i

Hence, expected total number of messages

$$= n + \sum_{i=1}^{n-1} \sum_{k=1}^{i} k \cdot P(i,k) \approx 0.69 n \log n + O(1)$$

Can we use fewer messages?

The $O(n^2)$ algorithm is simple and works in both synchronous and asynchronous model.

But can we solve the problem with fewer messages?

Idea:

Try to have msgs containing larger IDs travel smaller distance in the ring.

Hirschberg-Sinclair algorithm

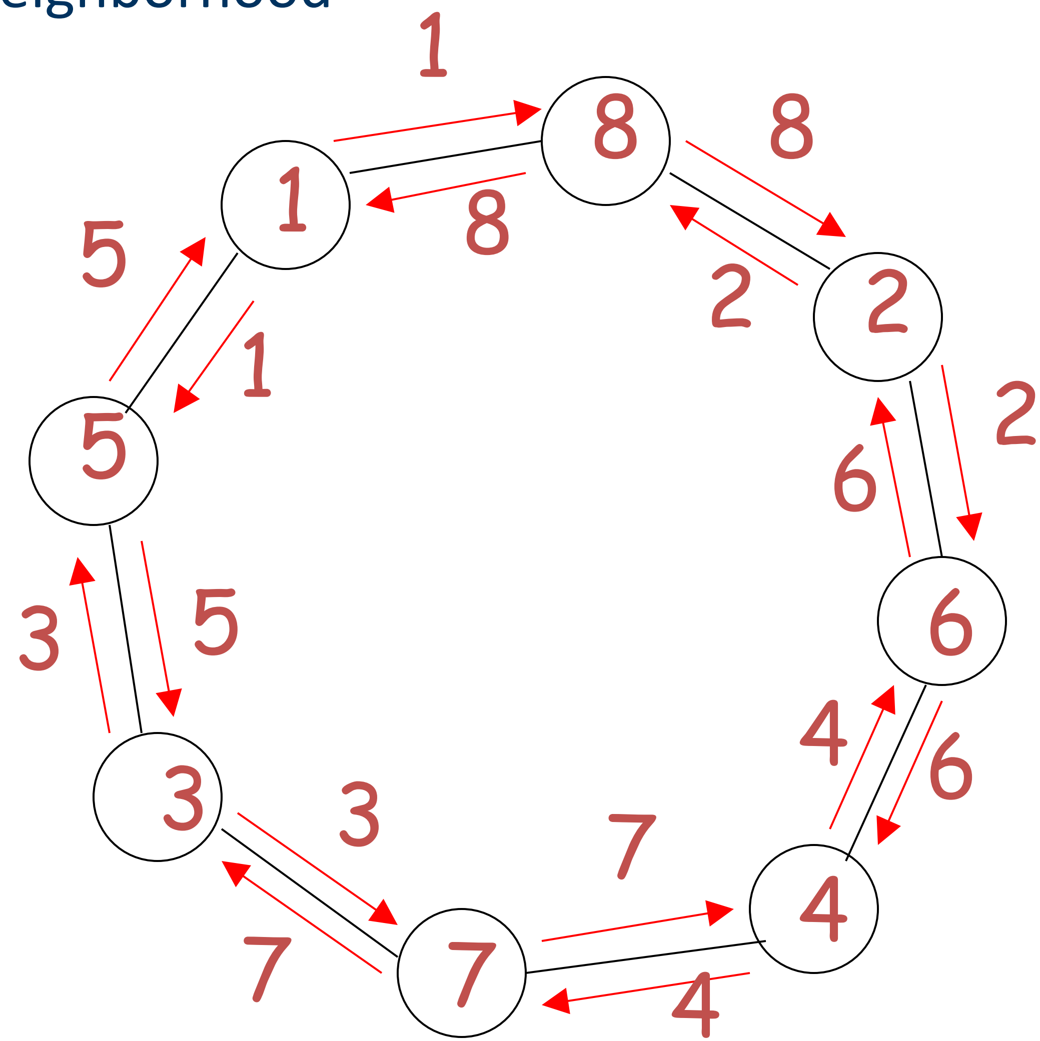
- Assume all nodes want to know the leader
- k-neighborhood of node p
- How does p send a message to distance k?
  - Message has a “time to live variable”
  - Each node decrements m.TTL on receiving
  - If m.TTL=0, don’t forward any more

Hirschberg-Sinclair algorithm (1)

- Algorithm operates in phases
- In phase 0, node p sends election message m to both p.NEXT and p.PREVIOUS with
  - m.ID=p.ID, and TTL=1
- Suppose q receives this message
  - Set m.TTL=0
  - If q.ID>m.ID, do nothing
  - If q.ID<m.ID, return message to p
- If p gets back both messages, it declares itself leader of its 1-neighborhood, and proceeds to next phase

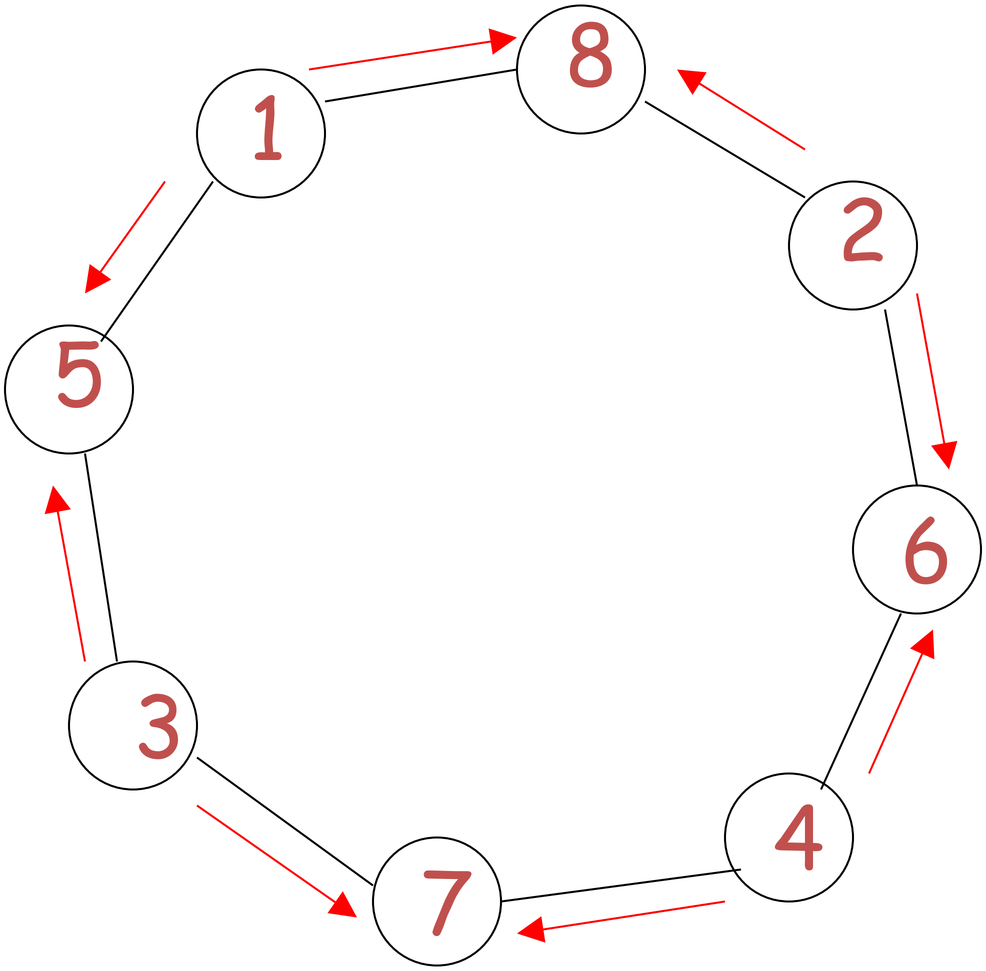
Hirschberg-Sinclair algorithm (1)

Phase 0: send(id, current phase, step counter)
to 1-neighborhood



Hirschberg-Sinclair algorithm (1)

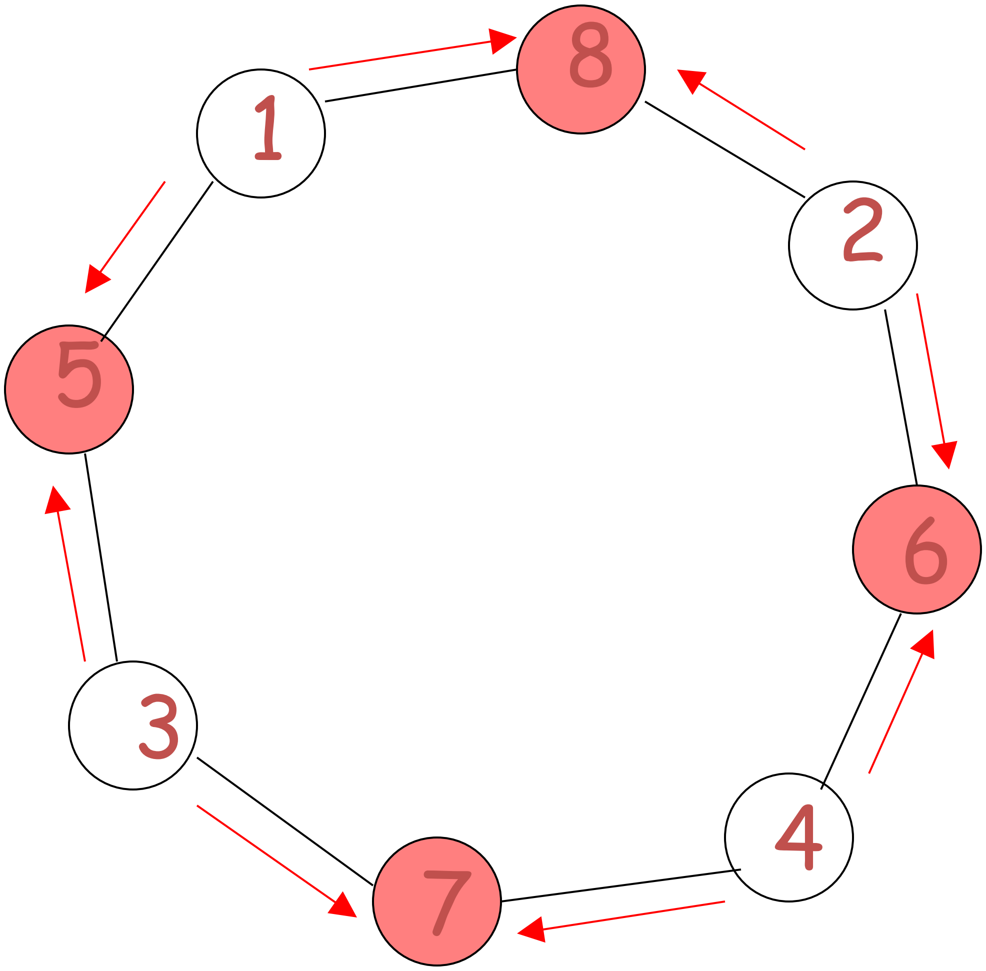
If: received ID>current ID
Then: send a reply(OK)



Hirschberg-Sinclair algorithm (1)

If: a node receives both replies

Then: it becomes a temporal leader & proceed to next phase



Hirschberg-Sinclair algorithm (2)

- Algorithm operates in phases
- In phase i, node p sends election message m to both p.NEXT and p.PREVIOUS with
    - m.ID=p.ID, and TTL=2^i

- Suppose q receives this message (from next/previous)
    - If m.TTL=0 then forward suitably to previous/next
    - Set m.TTL=m.TTL-1
    - If q.ID>m.ID, do nothing
    - ELSE:
        - If m.TTL=0 then return to sending process
        - else forward to suitably to previous/next
- If p gets back both messages, it is the leader of its 2^i neighborhood, and proceeds to phase $i + 1$

Hirschberg-Sinclair algorithm

- When $2^i \geq n/2$, only 1 processor survives & Leader found

- Number of phases: $O(\log n)$.

Question

What is the message complexity?

H&S algorithm: Message complexity

Question

What is the message complexity?

- In phase i
  - At most one node initiates message in any sequence of 2^{i-1} nodes
  - So, $n/2^{i-1}$ candidates
    - Each sends 2 messages, going at most 2^i distance, and transfers

      $2\times2\times2^i$ messages in total
  - $O(n)$ messages in phase i, and there are $O(\log n)$ phases
  - Total of $O(n\log n)$ messages.

H&S algorithm: Message complexity

Max # messages per leader		Max # current leaders
Phase 1:	4	n
Phase 2:	8	$n/2$
…		
Phase i:	2^{i+1}	$n/2^{i-1}$
…		
Phase log n:	$2^{\log n+1}$	$n/2^{\log n-1}$

H&S algorithm: Message complexity

Max # messages per leader		Max # current leaders	
Phase 1: 4	$\times$	n	$= 4n$
Phase 2: 8	$\times$	$n/2$	$= 4n$
…			
Phase i: 2^{i+1}	$\times$	$n/2^{i-1}$	$= 4n$
…			
Phase log n: $2^{\log n+1}$	$\times$	$n/2^{\log n-1}$	$= 4n$

Total messages: $O(n \cdot \log n)$

Can we go better?

- The $O(n\log n)$ algorithm is more complicated than the $O(n^2)$ algorithm but uses fewer messages in worst case.
- Works in both asynchronous case.

Can we reduce the number of messages even more?

Not in the asynchronous model.

Theorem

Any asynchronous Leader Election algorithm requires $\Omega(n\log n)$ messages.

Theorem

$\mathrm{O}(n)$ message synchronous algorithm exists for non-uniform ring.

Failures

Question

How do we know that something has failed?

Let's see what we mean by failed

Models of failure:

1. Assume no failures
2. Crash failures: Process may fail/crash
3. Message failures: Messages may get dropped
4. Link failures: a communication stops working
5. Some combinations of 2,3,4
6. Arbitrary failures: computation/communication may be erroneous

Failure detectors

- Detection of a crashed process
- A major challenge in distributed systems.
- A failure detector is a process that responds to questions asking whether a given processor has failed
  - A failure detector is not necessarily accurate

Failure detectors

- Reliable failure detectors
  - Replies with “working” or “failed”
- Difficulty:
  - Detecting something is working is easy: if they respond to a message, they are working
  - Detecting failure is harder: if they don’t respond to the message, the message may have been lost/delayed, maybe the processor is busy, etc.
- Unreliable failure detector
  - Replies with “suspected (failed)” or “unsuspected”
  - That is, does not try to give a confirmed answer
- We would ideally like reliable detectors, but unreliable ones (that say give “maybe” answers) could be more realistic

Simple example

- Suppose we know all messages are delivered within D seconds
- Then we can require each processor to send a message every T seconds to the failure detector
- If a failure detector does not get a message from process p in $T + D$ seconds, it marks p as "suspected" or "failed"

Synchronous vs asynchronous

- In a synchronous system there is a bound on message delivery time (and clock drift)
- So this simple method gives a reliable failure detector
- In fact, it is possible to implement this simply as a function
  - Send a message to process p, wait for $2D + \epsilon$ time
  - A dedicated detector process is not necessary
- In asynchronous systems, things are much harder

Simple failure detector

- If we choose T or D too large, then it will take a long time for failure to be detected
- If we select T too small, it increases communication costs and puts too much burden on processes
- If we select D too small, then working processes may get labeled as failed/suspected

Assumptions and real world

- In reality, both synchronous and asynchronous are too rigid
- Real systems are fast, but sometimes messages can take a longer time than usual (But not indefinitely long)
- Messages usually get delivered, but sometimes not…