Coordination Algorithms:

Leader Election

Leader Election

Let G = (V,E) define the network topology. Each process **i** has a variable **L(i)** that defines the ***leader***. The goal is to reach a configuration, where

$\forall$ i,j $\in$ V : i,j are non-faulty ::

(1) L(i) $\in$ V **and**

(2) L(i) = L(j) **and**

(3) L(i) is non-faulty

Often reduces to ***maxima (or minima) finding problem***.

(if we ignore the failure detection part)

Leader Election

Difference between mutual exclusion & leader election

The similarity is in the phrase "at most one process." But,

Failure is not an issue in mutual exclusion, a new leader is elected only after the current leader fails.

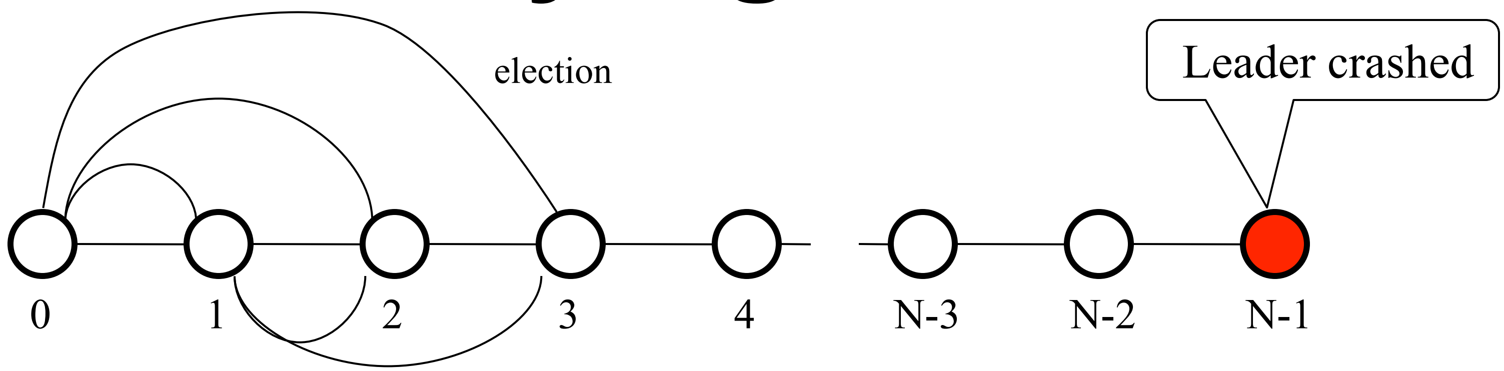
No fairness is necessary - it is not necessary that every aspiring process has to become a leader.

Bully algorithm

(Assumes that the topology is completely connected)

1. Send ***election*** message ***(I want to be the leader)*** to processes with ***larger id***
2. Give up your bid if a process with ***larger id*** sends a ***reply*** message ***(means no, you cannot be the leader).*** In that case, wait for the ***leader*** message (***I am the leader***). Otherwise elect yourself the leader and send a ***leader*** message
3. If ***no reply is received***, then elect yourself the leader, and broadcast a ***leader*** message.
4. If you receive a reply, but later don't receive a ***leader*** message from a process of larger id (i.e the leader-elect has crashed), then re-initiate election by sending ***election*** message.

Bully algorithm



Node 0 sends N-1 election messages
Node 1 sends N-2 election messages
Node N-2 sends 1 election messages etc

So, 0 starts all over again

Finally, node N-2 will be elected leader, but before it sent the leader message, it crashed.

The worst-case message complexity = **O(n^3) (This is bad)**

Maxima finding on a unidirectional ring

Chang-Roberts algorithm.

Initially all initiator processes are **red**.

Each initiator process i sends out ***token <i>***

{For each initiator i}

do token <j> received $\wedge$ j < i $\rightarrow$ skip (do nothing)

token <j>$\wedge$ j > i $\rightarrow$ send token <j>; color := **black**

token <j> $\wedge$ j = i $\rightarrow$ L(i) := I **{i becomes the leader}**

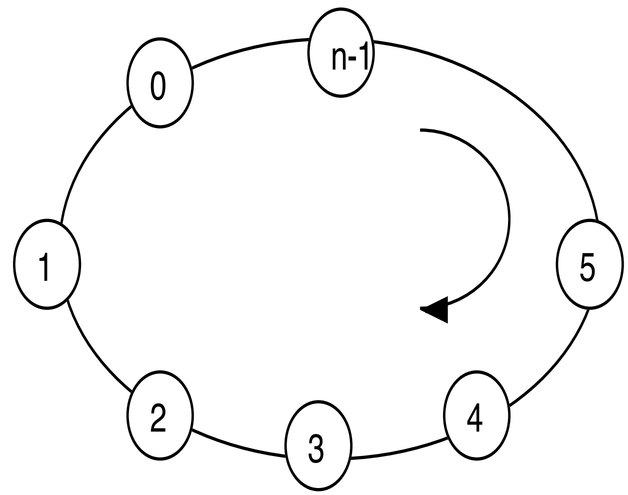
od

{Non-initiators remain **black**, and act as routers}

do token <j> received $\rightarrow$ send <j> **od**

Message complexity = O(n^2). Why?

What are the best and the worst cases?



The ids may not be nicely ordered like this

Bidirectional ring

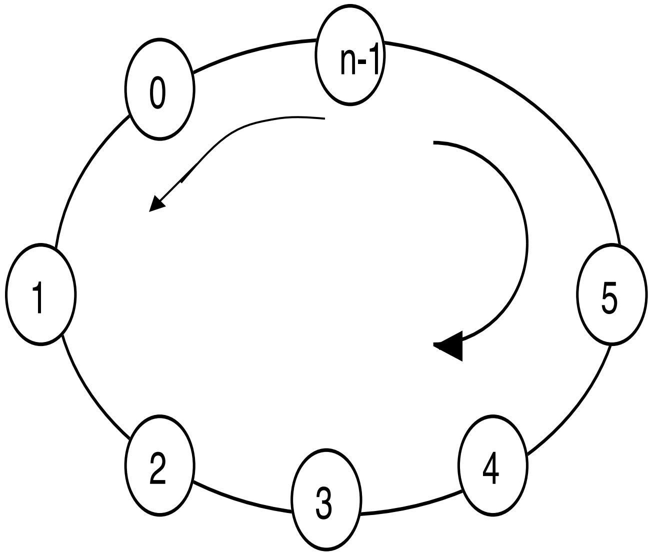
Franklin's algorithm (round based)

In each round, every process sends out *probes (same as tokens)* in **both** directions to its neighbors.

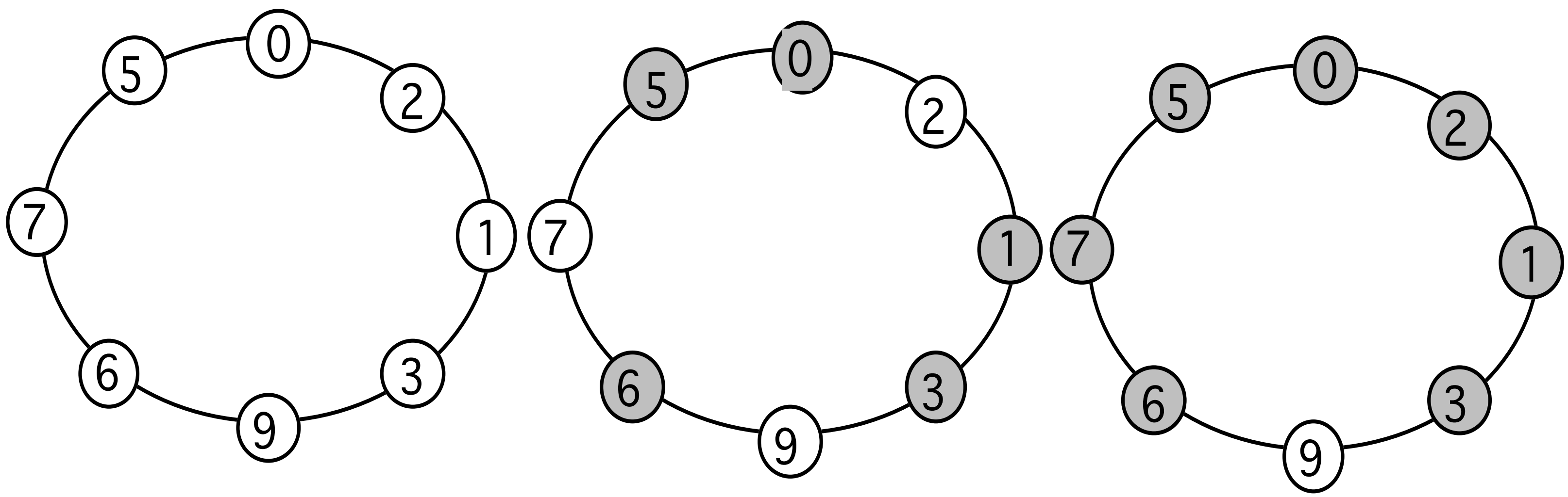
Probes from higher numbered processes will knock the lower numbered processes out of competition.

In each round, out of two neighbors, **at least one must quit**. So at least 1/2 of the current contenders will quit.

Message complexity = O(n log n). Why?



Sample execution



before round 0 after round 0 after round 1

Peterson's algorithm

initially $\forall$i : color(i) = **red**, alias(i) = **i**

{program for each round and for each red process}

send ***alias***; receive ***alias (N);***

if alias = alias (N) → **I am the leader**

alias ≠ alias (N) → send ***alias(N);*** receive ***alias(NN);***

if alias(N) > max (alias, alias (NN)) → alias:= alias (N)

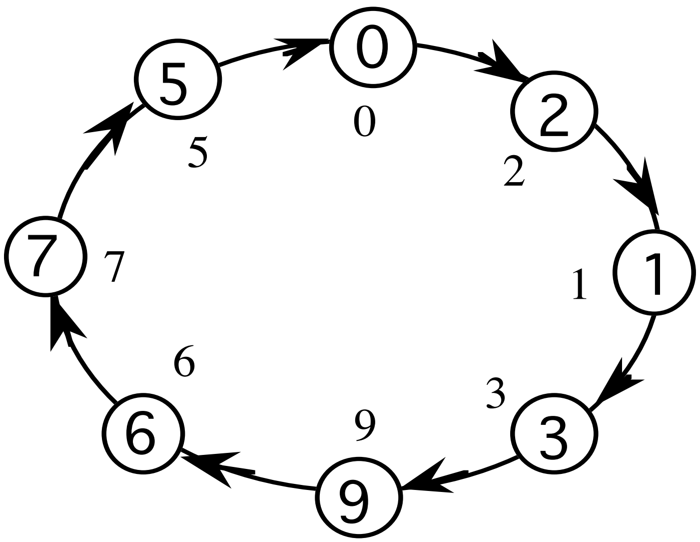
alias(N) < max (alias, alias (NN)) → color := **black**

fi

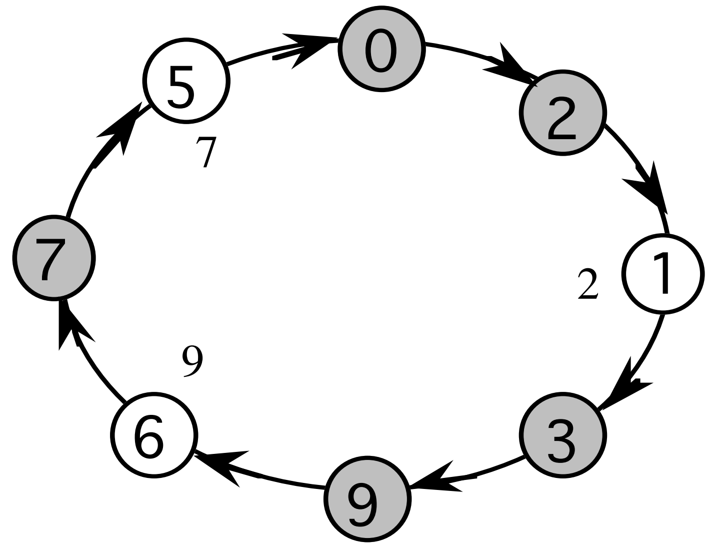
fi

{**N(i)** and **NN(i)** denote ***neighbor*** and ***neighbor's neighbor*** *of* ***i***}

Peterson’s algorithm



before round 0



after round 0

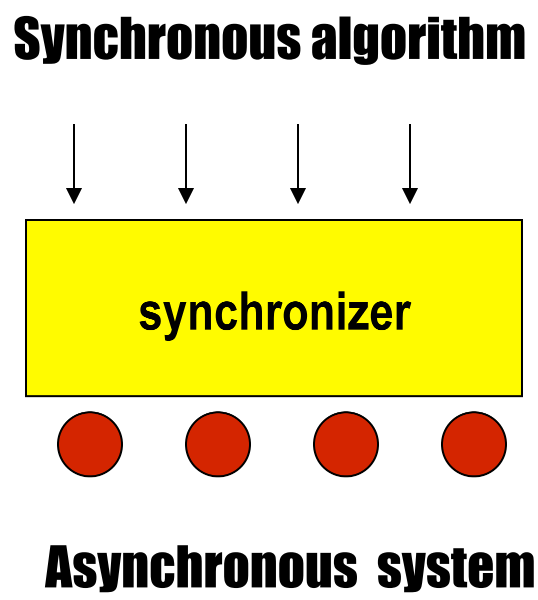
Round-based. Finds maxima on a unidirectional ring using O(n log n) messages. Uses an ***id*** and an ***alias*** for each process.

Synchronizers

Synchronous algorithms (round-based, where processes execute actions in lock-step synchrony) are easer to deal with than ***asynchronous algorithms***. In each round (or clock tick), a process

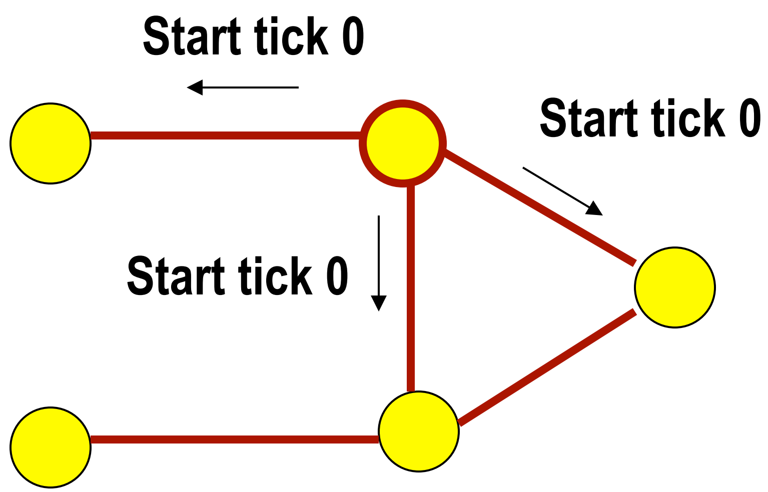
(1) receives messages from neighbors,
(2) performs local computation
(3) sends messages to ≥ 0 neighbors

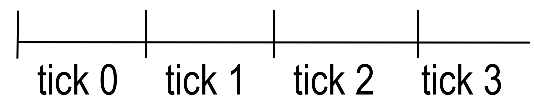
A **synchronizer** is a protocol that enables synchronous algorithms to run on an asynchronous system.



Synchronizers

"Every message sent in clock tick k must be received by the neighbors in the clock tick k." This is not automatic - some extra effort is needed. Consider a basic Asynchronous Bounded Delay (ABD) synchronizer



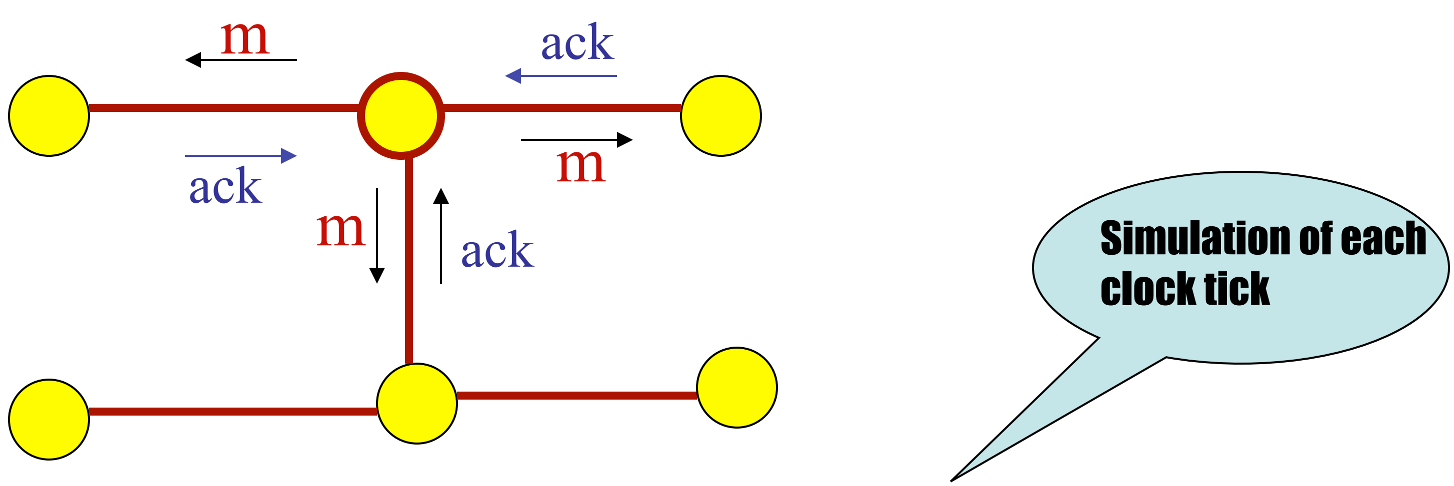


Channel delays have an upper bound d

Each process will ***start the simulation of a new clock tick after 2d time units***, where d is the maximum propagation delay of each channel

α-synchronizers

What if the propagation delay is arbitrarily large but finite?
The ***α-synchronizer*** *can handle this.*



1. Send and receive **messages** for the current tick.
2. Send **ack** for each incoming message, and receive **ack** for each outgoing message
3. Send a ***safe* message** to each neighbor after sending and receiving all **ack** messages (then follow steps 1-2-3-1-2-3- …)

Complexity of α-synchronizer

Message complexity $M(\alpha)$
Defined as the number of messages passed around the *entire network* for the simulation of each clock tick.

$M(\alpha) = O(|E|)$

Time complexity $T(\alpha)$
Defined as the number of *asynchronous rounds* needed for the simulation of each clock tick.

$T(\alpha) = 3$
(since each process exchanges m, ack, safe)

Complexity of α-synchronizer

$$M_A = M_S + T_S .\ M(\alpha)$$

MESSAGE complexity of the algorithm implemented on top of the asynchronous platform

Message complexity of the original synchronous algorithm

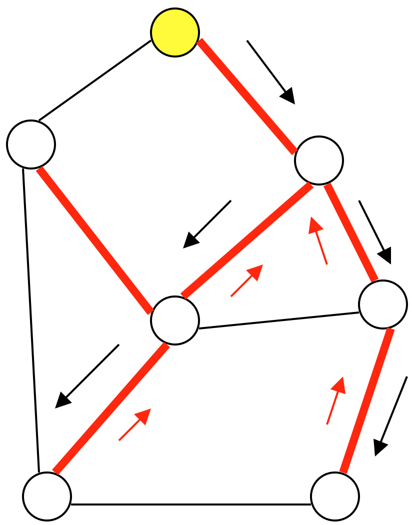
Time complexity of the original synchronous algorithm in rounds

$$T_A = T_S .\ T(\alpha)$$

TIME complexity of the algorithm implemented on top of the asynchronous platform

Time complexity of the original synchronous algorithm

The β-synchronizer



Form a *spanning tree* with any node as the root. The **root** initiates the simulation of each tick by sending message m(j) for each clock tick j. Each process responds with ack(j) and then with a safe(j) message **along the tree edges** (that represents the fact that the entire subtree under it is safe). When the root receives safe(j) from every child, it initiates the simulation of clock tick (j+1) using a next message.

To compute the message complexity M(β), note that
in each simulated tick, there are m messages of the original algorithm, m acks, and (N-1) safe messages and (N-1) next messages along the tree edges.

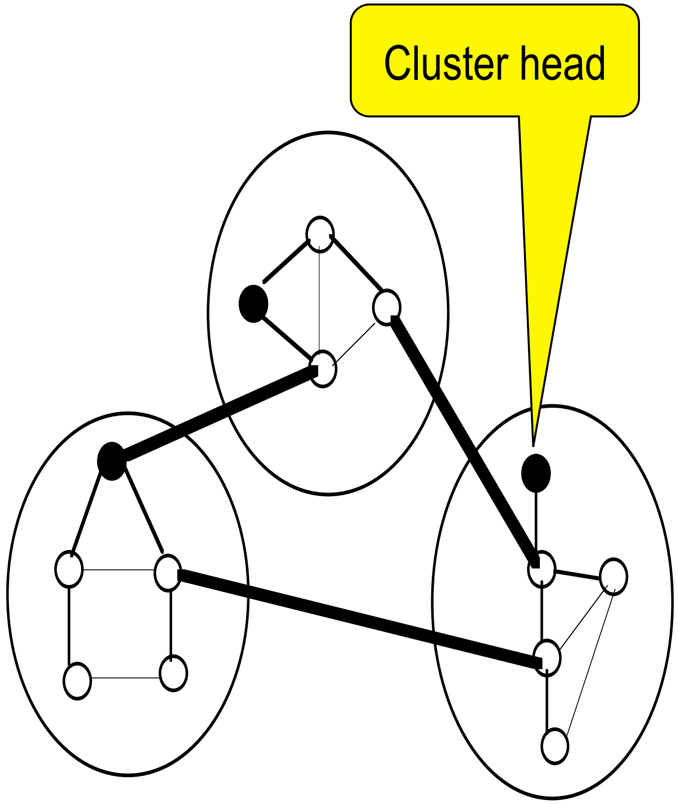
Time complexity T(β) = depth of the tree.
For a balanced tree, this is O(log N)

γ-synchronizer

Uses the best features of both α and β synchronizers. *(What are these?)**

The network is viewed as a tree of clusters. Each cluster has a cluster-head Within each cluster, β-synchronizers are used, but for inter-cluster synchronization, α-synchronizer is used

Preprocessing overhead for cluster formation.
The number and the size of the clusters is a crucial issue in reducing the message and time complexities



* α-synch has lower time complexity, β-synchronizers have lower message complexity

Example of application: Shortest path

- Consider Synchronous Bellman-Ford:
    - O(n |E|) messages, O(n) rounds
  - Asynchronous Bellman-Ford
    - Many corrections possible (exponential), due to message delays.
    - Message complexity exponential in n.
    - Time complexity exponential in n, counting message pileups.

- Using (e.g.) Synchronizer α:
    - Behaves like Synchronous Bellman-Ford.
    - Avoids corrections due to message delays.
    - Still has corrections due to low-cost high-hop-count paths.
    - O(n |E|) messages, O(n) time
    - Big improvement.