Distributed Coordination

Leader Election

Gerard LeLann posed the Election problem in a famous paper

Many distributed systems are client server based, with one server/coordinator (*leader*), and multiple clients

What happens if the leader fails?

- Elect a new one

Let $G = (V,E)$ define the network topology.
Each process **i** has a variable **L(i)** that defines the ***leader***.

$\forall$ i,j $\in$ V : i,j are non-faulty :: L(i) $\in$ V and
L(i) = L(j) and
L(i) is non-faulty

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

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 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.

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

Bully Algorithm

program Bully Algorithm
define
failed : boolean {true if current leader fails}
L : process {identifies leader}
m : message of election, leader, reply
state : idle, wait_leader, wait_reply
initially state = idle {for every process}

1. do failed -> $\forall$ j : j > i send election to j;
state := wait_reply;
failed :=false

2.(state=idle) AND (m=election) -> send reply to sender;
failed := TRUE

3. (state=wait_reply) AND (m=reply) ->
state :=wait for leader

Bully Algorithm

4. (state = wait_reply) AND timeout

   -> L(i) := i;

   ∀ j : j > i :: send leader to j;

   state := idle

5. (state = wait_leader) AND (m = leader)

   -> L(i) :=sender;

   state :=idle

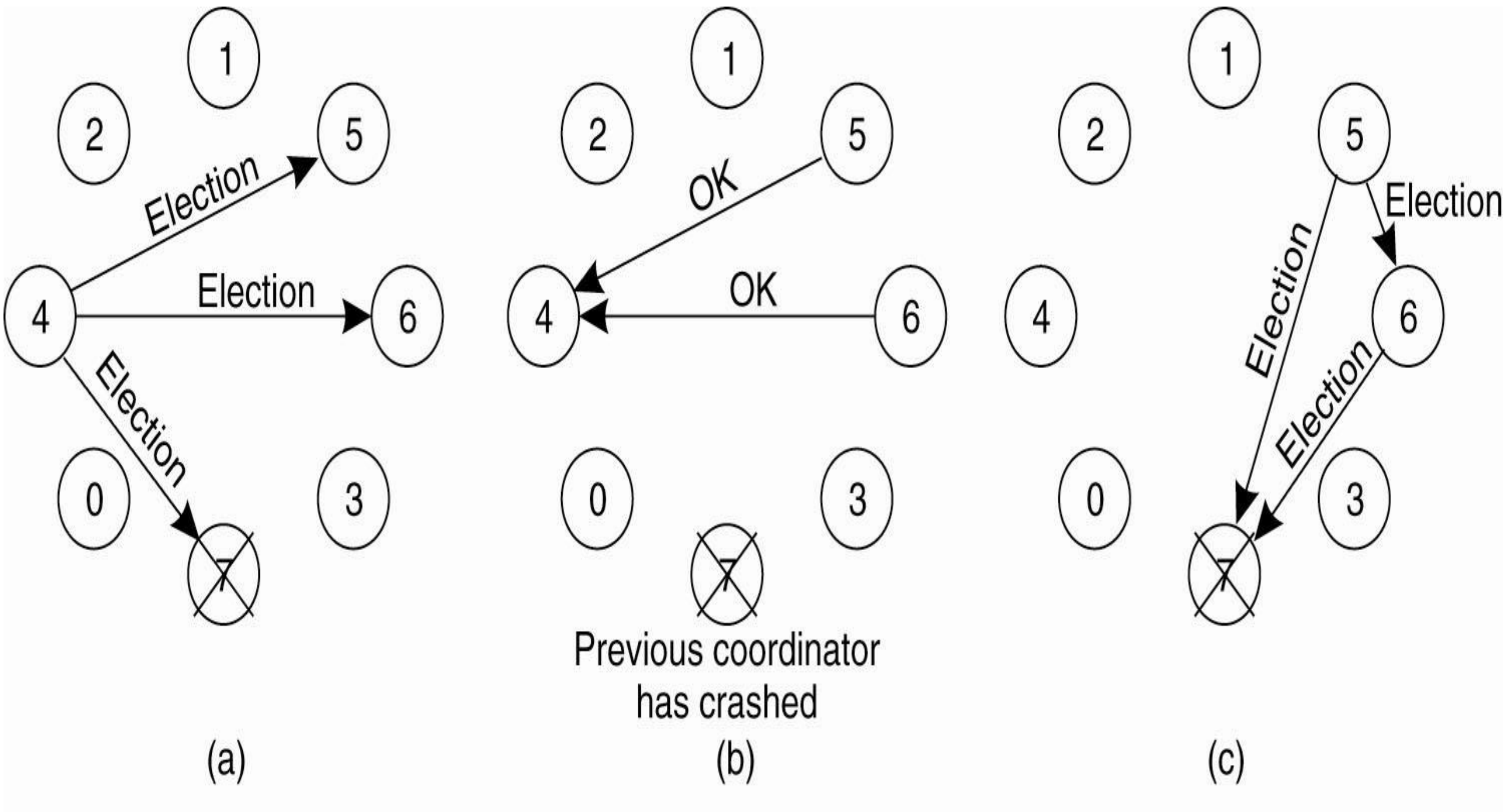
6. (state = wait_leader) AND (timeout)

   -> failed:= true;

   state :=idle

od

Bully Algorithm (Tanenbaum)



What about elections on rings?

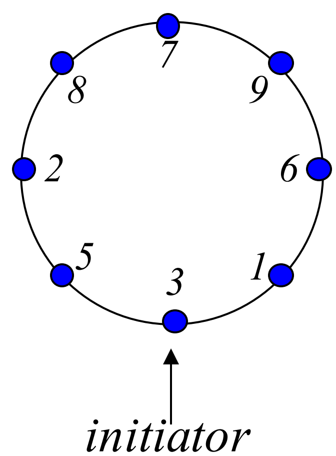
The first ever election algorithm was done for rings by LeLann

Simple idea:

- Let every initiator send a token with their identity around the whole ring
- Nodes are ***not*** allowed to initiate after they receive a token (example…)

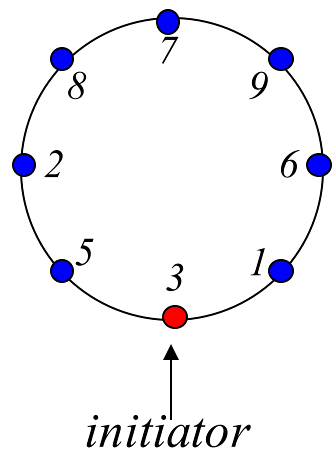
Example

- Nodes are ***not*** allowed to initiate a message after they receive a token (example…)



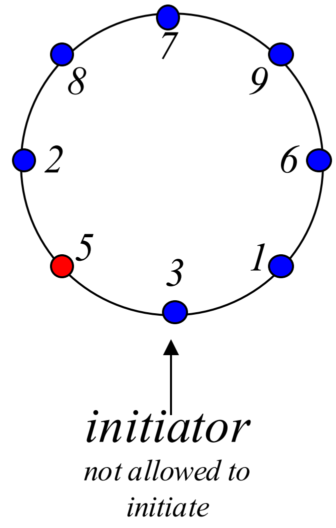
Example

- Nodes are ***not*** allowed to initiate a message after they receive a token (example…)



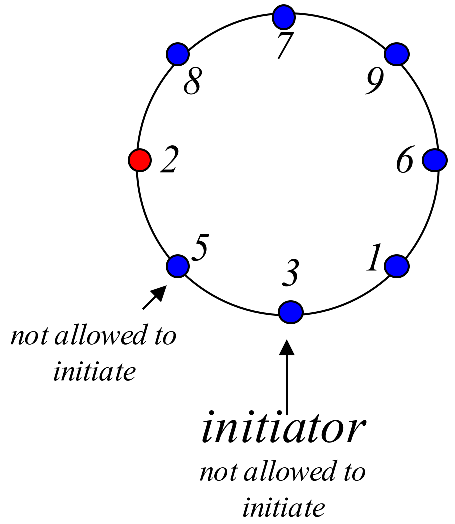
Example

- Nodes are ***not*** allowed to initiate a message after they receive a token (example…)



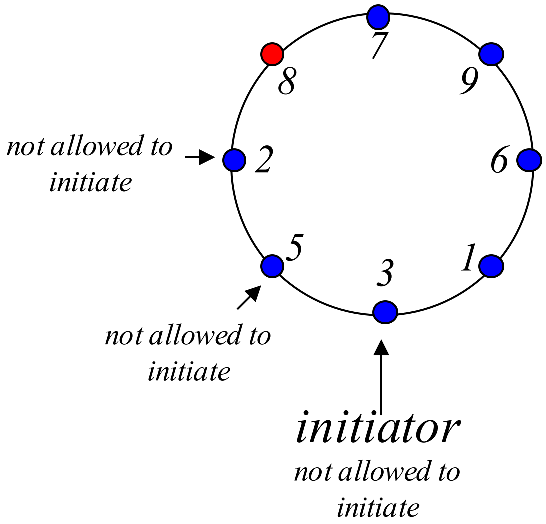
Example

- Nodes are ***not*** allowed to initiate a message after they receive a token (example…)



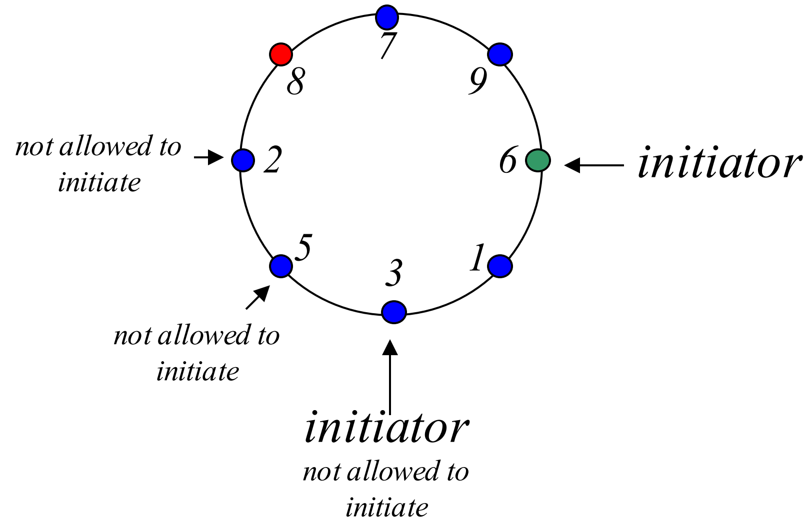
Example

- Nodes are ***not*** allowed to initiate a message after they receive a token (example…)



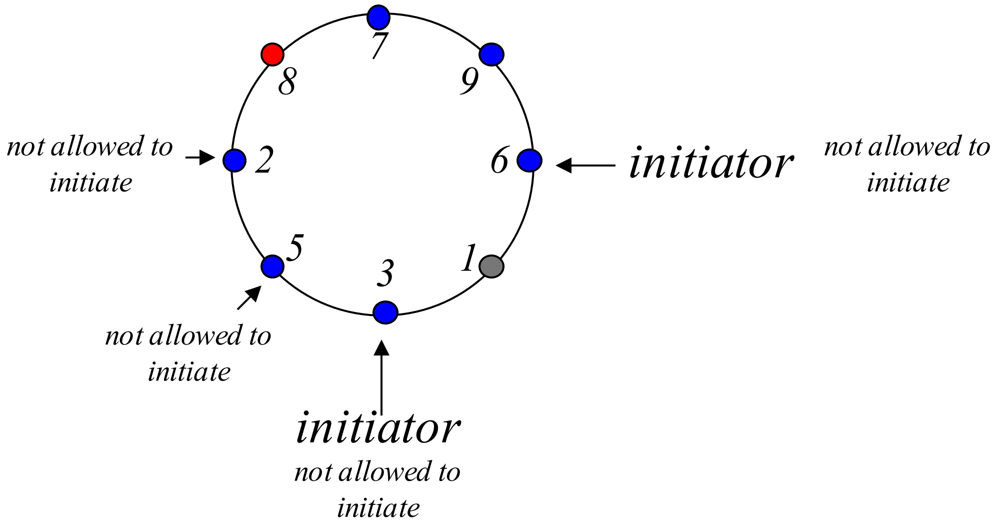
Example

- Nodes are ***not*** allowed to initiate a message after they receive a token (example…)



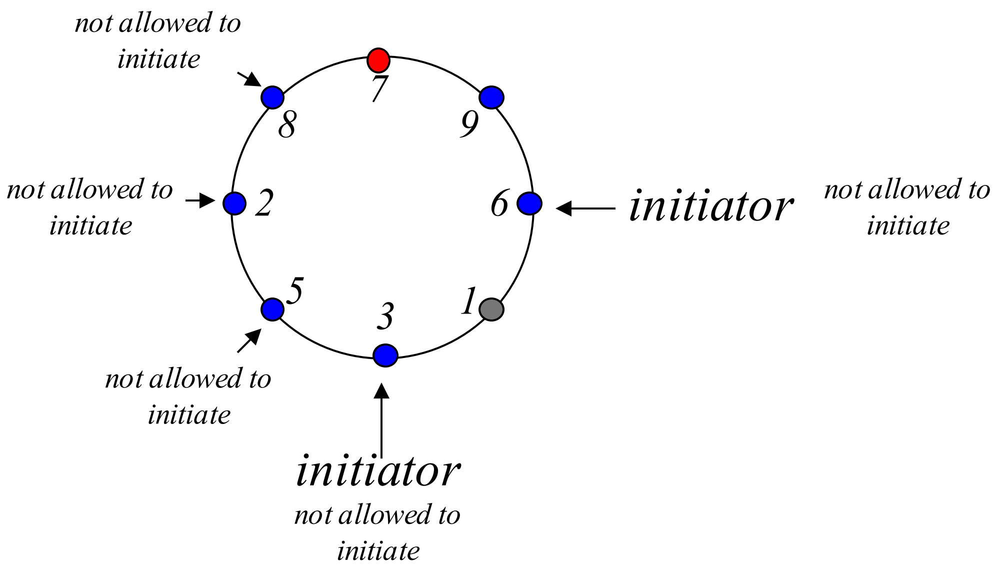
Example

- Nodes are ***not*** allowed to initiate a message after they receive a token (example…)



Example

- Nodes are ***not*** allowed to initiate a message after they receive a token (example…)



LeLann's ring election

Whenever a node receives back its id, it has seen every other initiators id

- Assuming FIFO channels

Let every node keep a list of every identifier seen ($list_p$)

- If non-initiator, $state=lost$ immediately
- If initiator, when own id received:
  - $state=leader$ if $min\{list_p\}=p$
  - $state=lost$ otherwise

LeLann's Algorithm

$$
\begin{array}{l}
\textbf{var } List_p \quad : \text{ set of } \mathcal{P} \quad \textbf{init } \{p\} ; \\
\quad state_p ; \\
\\
\textbf{begin if } p \text{ is initiator } \textbf{then} \\
\quad \textbf{begin } state_p := cand \text{ ; send } \langle \textbf{tok}, p \rangle \text{ to } Next_p \text{ ; receive } \langle \textbf{tok}, q \rangle \text{ ;} \\
\quad\quad \textbf{while } q \neq p \textbf{ do} \\
\quad\quad\quad \textbf{begin } List_p := List_p \cup \{q\} \text{ ;} \\
\quad\quad\quad\quad \text{send } \langle \textbf{tok}, q \rangle \text{ to } Next_p \text{ ; receive } \langle \textbf{tok}, q \rangle \\
\quad\quad\quad \textbf{end} \text{ ;} \\
\quad\quad \textbf{if } p = \min(List_p) \textbf{ then } state_p := leader \\
\quad\quad\quad\quad \textbf{else } state_p := lost \\
\quad \textbf{end} \\
\quad \textbf{else while } true \textbf{ do} \\
\quad\quad \textbf{begin } \text{receive } \langle \textbf{tok}, q \rangle \text{ ; send } \langle \textbf{tok}, q \rangle \text{ to } Next_p \text{ ;} \\
\quad\quad\quad \textbf{if } state_p = sleep \textbf{ then } state_p := lost \\
\quad\quad \textbf{end} \\
\textbf{end}
\end{array}
$$

Initially only know myself

Send my id, and wait

Repeat forwarding and collecting ids until we receive our id

***Termination**: did we win or lose*

Non-initiators just forward and lose

Message Complexity

Worst case is every node is initiator (N)

- Every initiator sends N messages

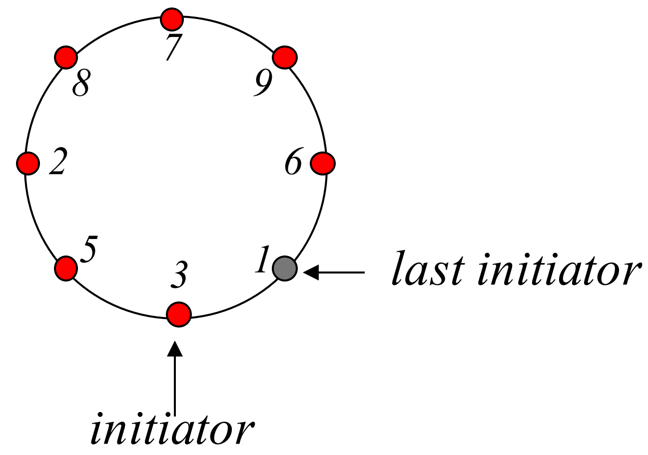
Gives a total of N^2 messages

Time Complexity

Assume last initiator f starts at time N-1

- f terminates after its token has circulated the ring, N steps

Time complexity $2N$-1



Chang-Roberts – An improvement

Chang and Roberts came up with a small improvement

Idea:

- When a node receives a token with smaller id than itself, why should it keep forwarding it?
- It is a waste, we know that that id will never win!
- Lets drop tokens with smaller ids than ourselves!

Chang Roberts Algorithm : Idea

1. Every process sends an *election* message with its id to the left process

2. if it has not seen a message from a higher process
 Forward any message with an id greater than own id to the left

3. If a process receives its own *election* message it is the leader

4. It then declares itself to be the leader by sending a *leader* message

Discussion about election

Are election algorithms of this "type" really useful?

- If a node in the ring breaks down, the circle is broken, election will not work (same for trees)
- The authors assume some external "connector" will fix the ring, and everything will proceed
  - Valid assumption?
  - In the case of a tree, if we anyway have to construct a new tree, the we could already embed leader information in that process
- Is it reasonable to assume that nodes have ordered finite set of ids?

Chang Roberts Algorithm

Chang-Roberts algorithm.

Initially all initiator processes are **red**.

{For each initiator i}
do token <j> $\wedge$ j > i $\rightarrow$ skip {j's token is removed}

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

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

od

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

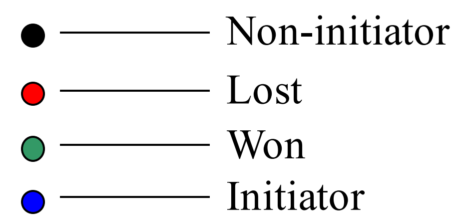
{for a non-initiator process}
do
token <j> received $\rightarrow$ color := black; send <j>
od

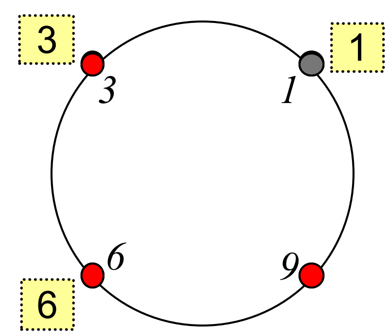
Chang Roberts - Example

Nodes 1, 3, 6 are initiators

```
var state_p ;

begin if p is initiator then
        begin state_p := cand ; send ⟨tok, p⟩ to Next_p ;
              while state_p ≠ leader do
                    begin receive ⟨tok, q⟩ ;
                          if q = p then state_p := leader
                          else if q < p then
                                begin if state_p = cand then state_p := lost ;
                                      send ⟨tok, q⟩ to Next_p
                                end
                    end
        end
     else while true do
              begin receive ⟨tok, q⟩ ; send ⟨tok, q⟩ to Next_p ;
                    if state_p = sleep then state_p := lost
              end
end
```



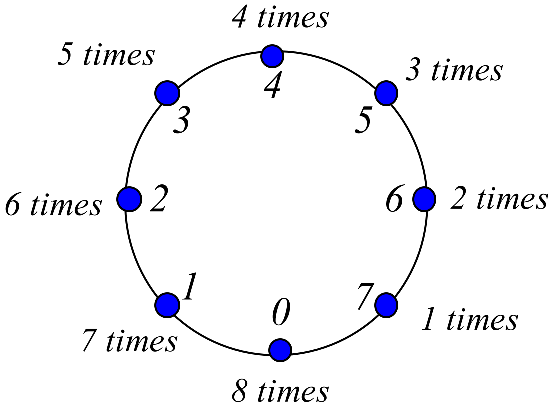


Chang Roberts Analysis

Worst case complexity same as LeLann's

- Time Complexity: *2N-1*
- Message Complexity: $O(N^2)$
  - Considered a sorted ring with N initiators

$$\sum_{i=0}^{N-1}(N-i) = N - \sum_{i=0}^{N-1} i = N - \frac{(N-1)N}{2} = \frac{(N+1)N}{2}$$



Synchronizers

Simulate a synchronous network over an asynchronous underlying network

Possible in the absence of failures

Enables us to use simple synchronous algorithms even when the underlying network is asynchronous

Synchronous network abstraction: *A message sent in pulse* i is received at pulse i+1

Synchronizer indicates when a process can generate a pulse

A process can go from pulse i to i+1 only when it has received and acted on all messages sent during pulse i-1

Synchronizers

In each pulse:

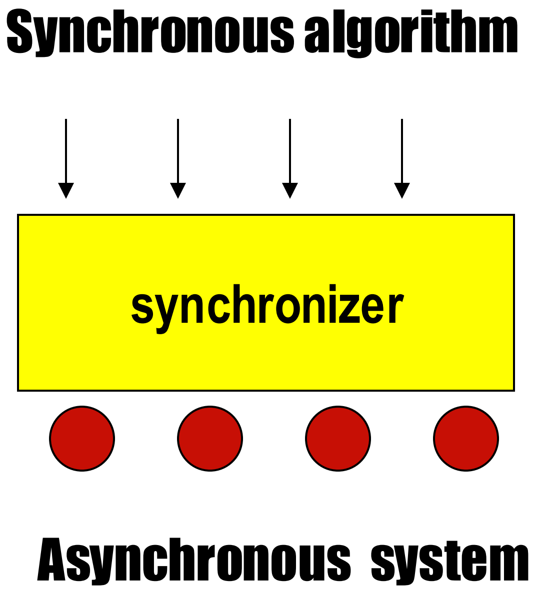
- A process receives messages sent during the previous pulse
- It then performs internal computation and sends out messages if required
- It can execute the next pulse only when the synchronizer permits it

Synchronizers

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

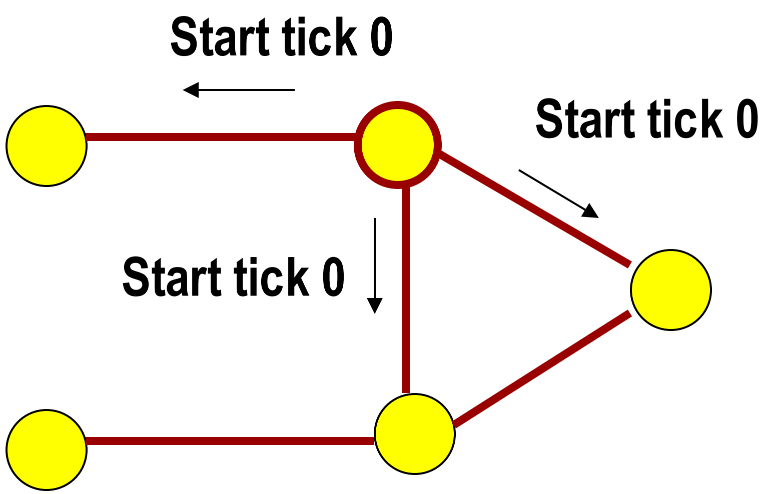
(1) receives messages from neighbors,
(2) performs local computation
(3) sends messages to ≥ 0 neighbors

A synchronizer is a protocol that enables synchronous algorithms to run on asynchronous platforms



Synchronizers

"Every message sent in clock tick k must be received by the receivers in the clock tick k." This is not automatic - some extra effort is needed.

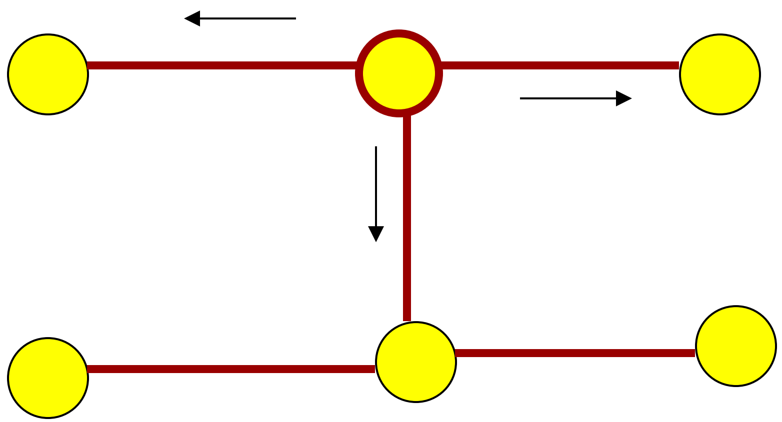


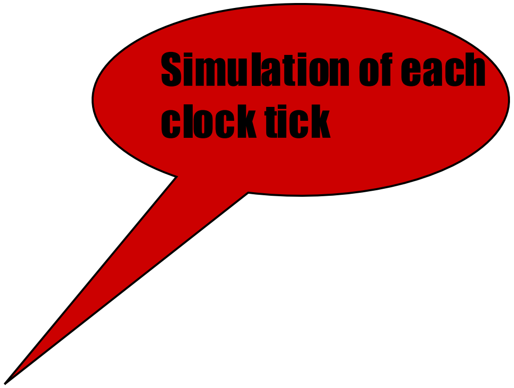
Asynchronous Bounded Delay (ABD) Synchronizer

In an ABD synchronizer, 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

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

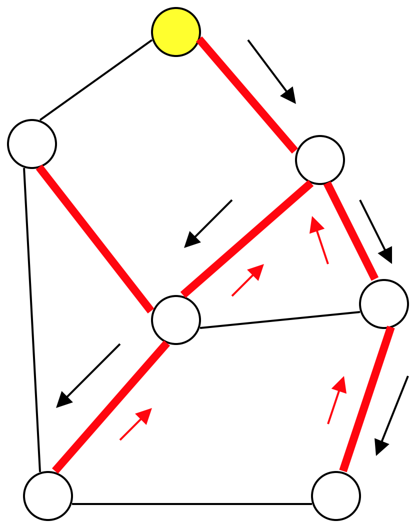
Message complexity of the original synchronous algorithm

Time complexity of the original synchronous algorithm

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

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

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 down the tree. Each process responds with ack(j) and then with a safe(j) message (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)

Message complexity M(β) = 3 (N-1)
since three messages (m, ack, safe) flow along each edge of the tree.

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

Implementation : Synchronizer interface

```
public interface Synchronizer {
        public void initialize();
             // initialize the synchronizer

      public void sendMessage(int destId, String tag,
                                  int msg);

        public void nextPulse();
             // block for the next pulse
}
```

Synchronizers : Overhead

The synchronous algorithm requires T_{synch} time and M_{synch} messages

Total Message complexity :

- $M_{asynch} = M_{init}+M_{synch}+M_{pulse}*T_{synch}$

- $T_{asynch}= T_{init}+T_{pulse}*T_{synch}$

Here M_{pulse}/ T_{pulse} are the messages/time required to simulate one pulse

A simple synchronizer

Every process sends exactly one message to all neighbors in each pulse

Wait for one message from each neighbor before executing the next pulse

If the synchronous algorithm sends multiple messages in one round, pack all messages together

If the synchronous algorithm does not send a message, the synchronizer sends a null message

A Simple Synchronizer: Algorithm

P_j::

var

$pulse$: integer initially 0;

round i :

$pulse := pulse + 1$;

wait for exactly one message with $(pulse = i)$ from each neighbors;

simulate the round i of the synchronous algorithm;

send messages to all neighbors with $pulse$;

Simple Synchronizer : Overhead

Initializing:

- M_{init}=0
- T_{init} =D

Each pulse

- M_{pulse} = 2E
- T_{pulse} =1

Application: BFS tree construction

Simple algorithm

- *root* starts the computation, sends *invite* to all neighbors
- If P_i receives an invite for the first time (say from node P_j) then i sets j as its parent and sends invitations to all neighbors
- Ignore all successive *invites*

Application: BFS tree construction

This algorithm does not always give a BFS tree

Run it with a synchronizer to ensure that the BFS tree is computed

```java
//BFS tree with a synchronizer
public class SynchBfsTree extends Process {
    int parent = -1;    int level;
    Synchronizer s;     boolean isRoot;

 ...
    public void initiate() {
        if (isRoot) {
            parent = myId;
            level = 0;
        }
        s.initialize(this);
        for (int pulse = 0; pulse < N; pulse++) {
            if ((pulse == 0) && isRoot) {
                for (int i = 0; i < N; i++)
                    if (isNeighbor(i))
                        s.sendMessage(i, "invite", level + 1);
            }
            s.nextPulse();
        }
    }
    public void handleMsg(Msg m, int src, String tag) {
        if (tag.equals("invite")) {
            if (parent == -1) {
                parent = src;
                level = m.getMessageInt();
                Util.println(myId + " is at level " + level);
                for (int i = 0; i < N; i++)
                    if (isNeighbor(i) && (i != src))
                        s.sendMessage(i, "invite", level + 1);
}   }   }   }
```

Synchronizer α

Very similar to the Simple synchronizer

The synchronizer generates the next pulse if all the neighbors are safe

Inform all neighbors when you are safe.

Acknowledge all messages so that the sending process knows when it is safe

$T_{init} = D$ $M_{init} = D$

$T_{pulse} = O(1)$ $M_{pulse} = O(E)$

Synchronizer β

Idea: reduce the message complexity at the cost of time complexity
Assume the existence of a rooted spanning tree
A node sends *subtree-safe* when all the nodes in its subtree are safe
When the root receives *subtree-safe* from all children it broadcasts *pulse* (using the simple broadcast discussed in the last chapter)

Synchronizer β : Overhead

$T_{init} = O(N)$ $M_{init}=O(\ N\log N + E)$

$T_{pulse}= O(N)$ $M_{pulse}=O(N)$

Acknowledgements

This part is heavily dependent on the course : CS4231 Parallel and Distributed Algorithms, NUS by Dr. Haifeng Yu and Vijay Hargh Elements of Distributed Computing Book and Sukumor Ghosh Distributed Systems Course 22C:166 at Iowa University.