Modern Distributed Computing

Theory and Applications

Ioannis Chatzigiannakis

Sapienza University of Rome

Lecture 2
Tuesday, March 12, 2013

Part 1: Static Synchronous Networks

1. Synchronous Message-passing Model, Definitions
2. Anonymous Leader Election, Impossibility Results
3. Symmetry Breaking Algorithms, Randomization
4. Leader Election Algorithms
5. Broadcast, Convergecast
6. Lower Bounds

Modeling Processes

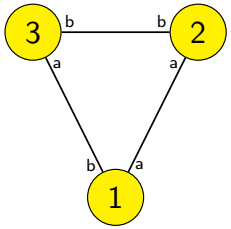
- The system is comprised from a collection of processing elements or "processors".
  - The "processing element" suggests a piece of hardware.
  - The "processors" suggests some kind of logical entity (i.e., software).
  - For simplicity we may assume that each processing element has 1 processor.
- Processors execute a collection of processes.
  - For simplicity we may assume that each processor executes only one process.
  - We also assume that each process can be executed by a single processor.

Modeling the Communication Network

- The processing elements (i.e., the processes) are connected via a connected network (i.e., there exists 1 path between any pair of processes).
- We define the network as a graph $G = (V, E)$:
  - comprised of a finite set V of points – the vertices – representing the processing untis (i.e., processes) – $n = |V|$
  - a collection E of ordered pairs of elements of V ($E \subset [V]^2$) – the edges – representing the communication channels of the network – $m = |E|$

Modeling Communication Channels

- Channels are the edges of the graph.
  - The edges may be directed – to represent unidirectional communication.
  - or undirected – to represent bidirectional communication.
- Processes can distinguish each communication channel and select a specific one to use.

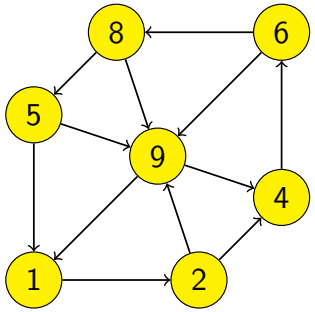


Modeling Messages

- Data exchange over communication channels is done via message exchanges.
- We assume that each communication channel may transmit only one message at any time instance.
- We assume that there exists a fixed message alphabet M
  - remains fixed throughout the execution of the system.
  - contains the symbol `null` a placeholder indicating the absence a message.

Neighboring Processes

- We say vertex v is outgoing neighbor of vertex u if
  - the edge uv is included in G.
- We say vertex u is incoming neighbor of vertex v if
  - the edge uv is included in G.
- We define $nbrs_u^{out} = \{v|(u,v) \in E\}$ all the vertices that are *outgoing neighbors* of vertex u.
- We define $nbrs_u^{in} = \{v|(v,u) \in E\}$ all the vertices that are *incoming neighbors* of vertex u.



5 is outgoing neighbor of 8
8 is incoming neighbor of 5
$nbrs_9^{out} = \{1, 4\}$
$nbrs_9^{in} = \{2, 5, 6, 8\}$

Network Properties

`distance(u,v)`

Let `distance(u,v)` denote the length of the shortest directed path from u to j in G, if any exists; otherwise `distance(u,v)`$=\infty$.

`diam(G)`

Let `diam(G)` denote the diameter of the graph G, the maximum distance `distance(u,v)`, taken over all paths (u, v).

Network Topology & Initial Knowledge

- Distributed algorithms may be designed for a specific network topology
  - ring, tree, fully connected graph ...
- Distributed algorithm may be designed for networks with specific properties
  - we say that the algorithm has "initial knowledge"
- An algorithm assuming a large number of specific properties is called "weak" algorithm.
  - An algorithm that does not assume any specific property is called "strong" algorithm – since it can be executed in a broader range of possible networks.

Process States

- Each process $u \in V$ is defined by a set of states $states_u$
  - A nonempty set of states $start_u$, known as starting states or initial states.
  - A nonempty set of states $halt_u$, known as halting states or terminating states.
- Each process uses a message-generator function $msgs_u : states_u \times nbrs_u^{out} \rightarrow M \cup \{\texttt{null}\}$
  - given a current state,
  - generates messages for each neighboring process.
- Uses a state-transition function $trans_u : states_u \times (M \cup \{\texttt{null}\})^{nbrs_u^{in}} \rightarrow states_u$
  - given a current state,
  - and messages received,
  - computes the next state of the process.

System Initialization

- Initially
  - all processes are set to an initial state,
  - all channels are empty.
- Algorithms groups processes in two sets
  1. Initiators – a process is initiator if it activates the execution of the algorithm in the local neighborhood.
  2. Non-initiators – a non-initiating process is activated when a message is received from a neighboring process.

Centralized vs Decentralized

An algorithm is classified as centralized if the exists one and only one initiator in each execution and decentralized if the algorithm may be initialized with an arbitrary subset of processes.

- Usually centralized algorithms achieve low message complexity.
- Usually decentralized algorithms achieve improved performance in the presence of failures.

Uniformity

An algorithm is uniform if its description is independent of the network size n.

- A property that holds for a small network size, also holds for large network sizes.
- We only have to examine the behavior of a protocol (for a given property) in small network sizes.

Algorithm execution: Steps and Rounds

- All processes, repeat in a "synchronized" manner the following steps:

  1^{st} Step

  1. Apply the message generator function.
  2. Generate messages for each outgoing neighbor.
  3. Transmit messages over the corresponding channels.

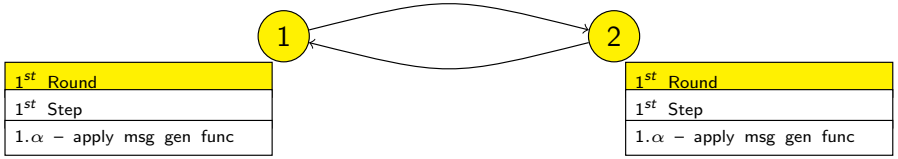
  2^{nd} Step

  1. Apply the state transition function.
  2. Remove all incoming messages from all channels.
- The combination of these two steps is called a round (of execution).

Example of execution of a Synchronous System

- Initially
  - all processes are set to an initial state,
  - all channels are empty.
- the processes execute in a "synchronized" manner the protocol.

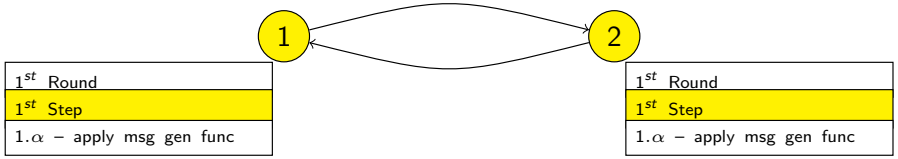
Execution of Synchronous System



Example of execution of a Synchronous System

- Initially
  - all processes are set to an initial state,
  - all channels are empty.
- the processes execute in a "synchronized" manner the protocol.

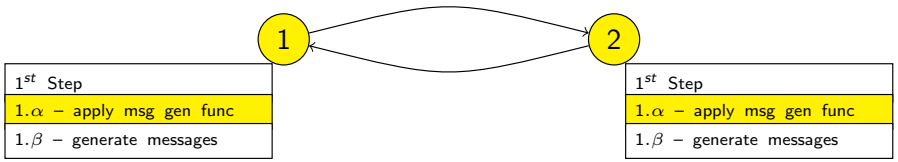
Execution of Synchronous System



Example of execution of a Synchronous System

- Initially
  - all processes are set to an initial state,
  - all channels are empty.
- the processes execute in a "synchronized" manner the protocol.

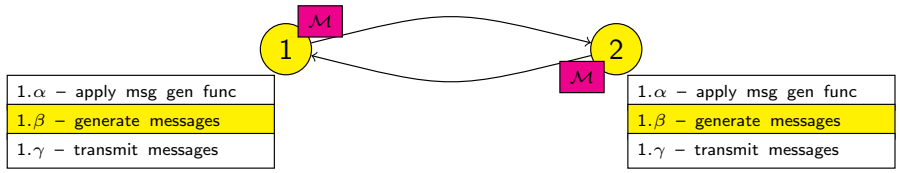
Execution of Synchronous System



Example of execution of a Synchronous System

- Initially
  - all processes are set to an initial state,
  - all channels are empty.
- the processes execute in a "synchronized" manner the protocol.

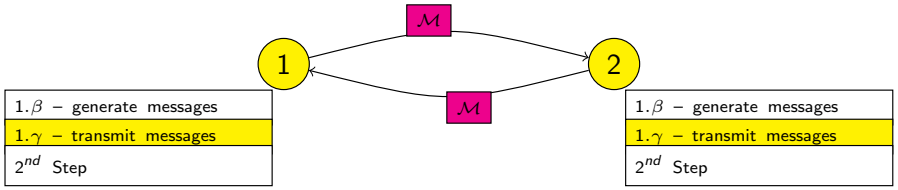
Execution of Synchronous System



Example of execution of a Synchronous System

- Initially
  - all processes are set to an initial state,
  - all channels are empty.
- the processes execute in a "synchronized" manner the protocol.

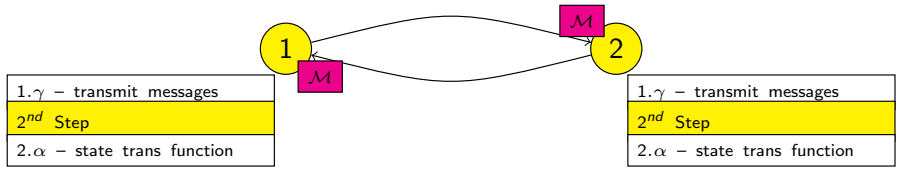
Execution of Synchronous System



Example of execution of a Synchronous System

- Initially
  - all processes are set to an initial state,
  - all channels are empty.
- the processes execute in a "synchronized" manner the protocol.

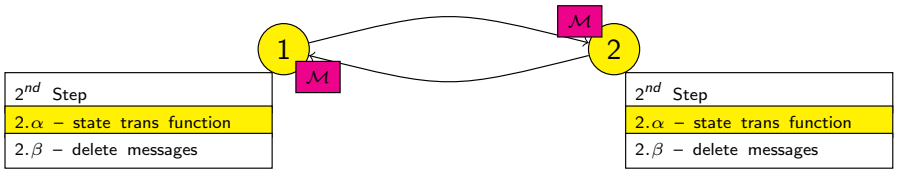
Execution of Synchronous System



Example of execution of a Synchronous System

- Initially
  - all processes are set to an initial state,
  - all channels are empty.
- the processes execute in a "synchronized" manner the protocol.

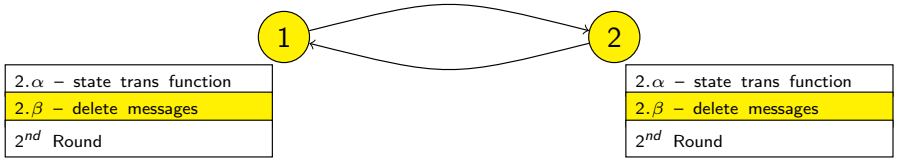
Execution of Synchronous System



Example of execution of a Synchronous System

- Initially
  - all processes are set to an initial state,
  - all channels are empty.
- the processes execute in a "synchronized" manner the protocol.

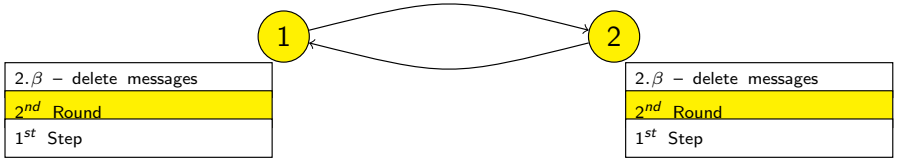
Execution of Synchronous System



Example of execution of a Synchronous System

- Initially
  - all processes are set to an initial state,
  - all channels are empty.
- the processes execute in a "synchronized" manner the protocol.

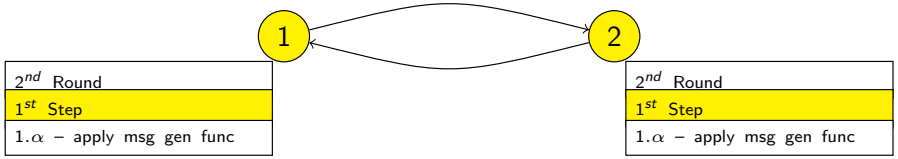
Execution of Synchronous System



Example of execution of a Synchronous System

- Initially
  - all processes are set to an initial state,
  - all channels are empty.
- the processes execute in a "synchronized" manner the protocol.

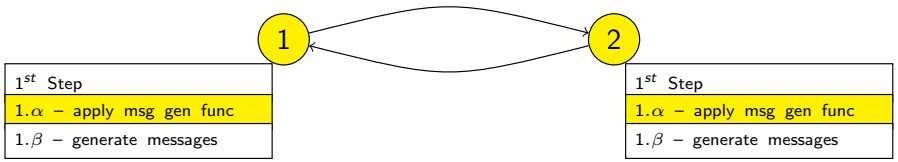
Execution of Synchronous System



Example of execution of a Synchronous System

- Initially
  - all processes are set to an initial state,
  - all channels are empty.
- the processes execute in a "synchronized" manner the protocol.

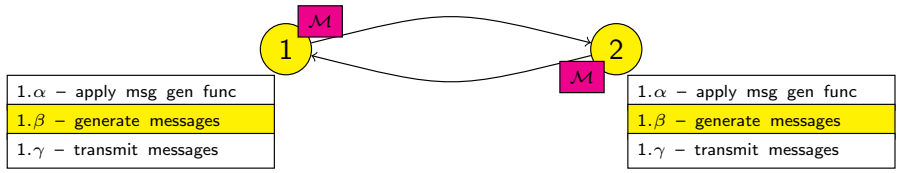
Execution of Synchronous System



Example of execution of a Synchronous System

- Initially
  - all processes are set to an initial state,
  - all channels are empty.
- the processes execute in a "synchronized" manner the protocol.

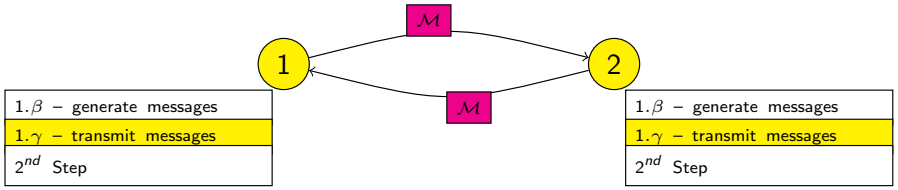
Execution of Synchronous System



Example of execution of a Synchronous System

- Initially
  - all processes are set to an initial state,
  - all channels are empty.
- the processes execute in a "synchronized" manner the protocol.

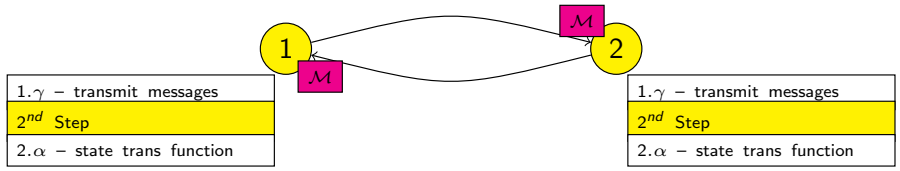
Execution of Synchronous System



Example of execution of a Synchronous System

- Initially
  - all processes are set to an initial state,
  - all channels are empty.
- the processes execute in a "synchronized" manner the protocol.

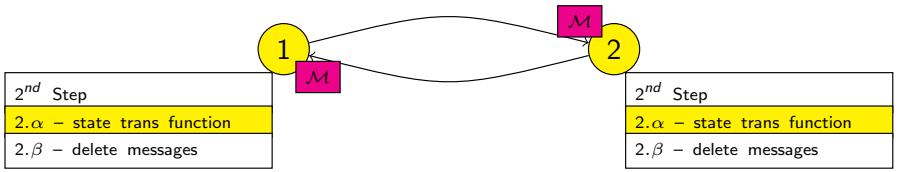
Execution of Synchronous System



Example of execution of a Synchronous System

- Initially
  - all processes are set to an initial state,
  - all channels are empty.
- the processes execute in a "synchronized" manner the protocol.

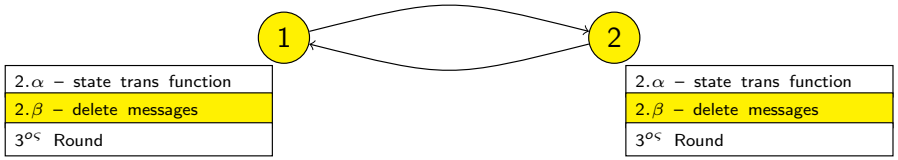
Execution of Synchronous System



Example of execution of a Synchronous System

- Initially
  - all processes are set to an initial state,
  - all channels are empty.
- the processes execute in a "synchronized" manner the protocol.

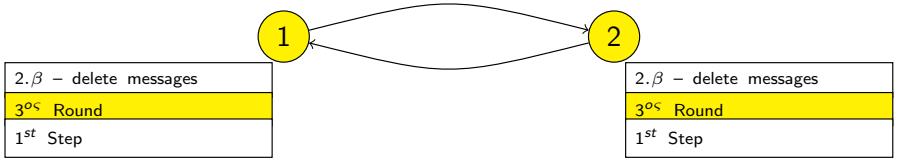
Execution of Synchronous System



Example of execution of a Synchronous System

- Initially
  - all processes are set to an initial state,
  - all channels are empty.
- the processes execute in a "synchronized" manner the protocol.

Execution of Synchronous System



System Configuration

We wish to describe the execution of a distributed algorithm.

- We assume a sequence of state transitions of the processes of the system
  - produced as result of transmissions and receptions of messages, or
  - internal (to each process) reasons.
- Lets assume a given time instance i
  - each process u is in state $states_u$.
  - the characterization of the state of all processes defines a configuration of the system C_i.

Execution of a distributed algorithm

- Initially, processes execute a single round of the algorithm
  - a given set of message transmissions M_i take place,
  - a given set of message N_i are received.
- The next round $i+1$, we say that the system is in configuration C_{i+1}
- The execution of the distributed algorithm can be defined as an infinite sequence
  $C_0, M_1, N_1, C_1, M_2, N_2, C_2, \ldots$

Basic Failure Types

- We define two abstract types of failures:
  1. failures occurring during the transmission of messages,
  2. failures occurring on the processing elements (processors).
- Communication failure: a failure during the transmission of a single message over a specific channel of the network.
- Stopping failure: a process terminates, either before, or after, or during the execution of some part of the 1^{st} or 2^{nd} step of the round.
  - A failure may happen during the generation of messages, therefore some outgoing messages are transmitted.

Byzantine Failures

- The network includes faulty processes that do not terminate but continue to participate in the execution of the algorithm.
- The behavior of the processes may be completely unpredictable.
- The internal state of a faulty process may change during the execution of a round arbitrarily, without receiving any message.
- A faulty process may send a message with any content (i.e., fake messages), independently of the instructions of the algorithm.
- We call such kind of failures as Byzantine failures.
- We use byzantine failures to model malicious behavior (e.g. cyber-security attacks).

Why study Byzantine Fault Tolerance?

- Does this happen in the real world?
  The "one in a million" case.
  - Malfunctioning hardware,
  - Buggy software,
  - Compromised system due to hackers.
- Assumptions are vulnerabilities.
- Is the cost worth it?
  - Hardware is always getting cheaper,
  - Protocols are getting more and more efficient.

Measuring Performance

- We wish to study the performance of the system.
  - We define the minimum requirement,
  - Select a suitable distributed algorithm.
- How can we measure performance?
- We use to fundamental metrics to define the complexity of distributed algorithms:
  1. Time complexity
  2. Communication complexity

Time Complexity

The time complexity of a synchronous system is defined as the total number of rounds required for all the processes to produce all the necessary output, or until all processes enter a halting state.

- Directly related with the execution time of an algorithm.
- In practice, the execution time of a distributed algorithm is the most important performance metric.

Communication Complexity

The communication complexity of a synchronous system is defined as the total number of non-null messages exchanged during the execution of the system.

- In some cases it is measured in total number of `bits` exchanged.
  - in cases when the volume of messages produces congestion in the network,
  - and the execution of the algorithm is delayed (for the network to deliver messages).

Communication Complexity

- In real conditions
  - multiple algorithms are executed concurrently,
  - they share the same communication medium.
  - What is the contribution of each algorithm to the total network congestion ?
- It is difficult to quantify the effect that the messages of each algorithm have on the performance of the other algorithms.
- In general, at design time, we always wish to minimize the messages produced by our algorithms.

Books & Seminal Papers

1. Nancy A Lynch: "Distributed Algorithms". Morgan Kaufmann (1996)
2. Michael J. Fischer, Nancy A. Lynch: "A Lower Bound for the Time to Assure Interactive Consistency". Inf. Process. Lett. 14(4): 183-186 (1982)
3. Harry R. Lewis, Christos H. Papadimitriou: "Elements of the Theory of Computation". Prentice Hall (1981)
4. Barbara Liskov, Alan Snyder, Russell R. Atkinson, Craig Schaffert: Abstraction Mechanisms in CLU. Commun. ACM 20(8): 564-576 (1977)

Leader Election

The election of a leader in a network requires the selection of a single, unique, process that will enter a state "leader" (or "elected") while all other processes enter the state "non-leader" (or "non-elected").

- The problem of leader election was formulated for the first time by LeLann (1977).
- The problem depicts the basic characteristics of a large family of problems encountered in real distributed systems.
- The problem has many variations.
- We start by considering the simple case where the network is a ring.

More than one of anything

- Redundancy is a common technique towards robustness against failures.
- This means that we sometime have a service installed on several machines for redundancy, but only one of the is active at any given moment.
- Yet, you cannot rely on manual monitoring of the redundant services whose presence is crucial to the overall health of the system.
- An automated approach is necessary for an active-passive redundancy of a singleton service in production environments.

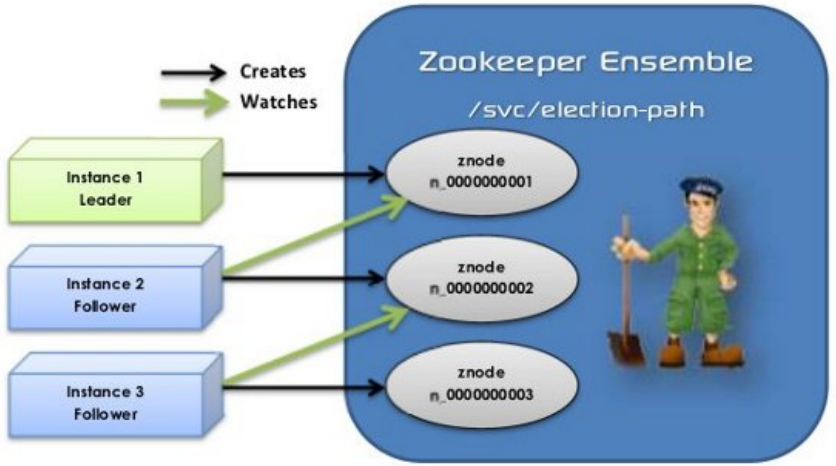
Yahoo Research / Apache Zookeeper

- One of the most prominent open source implementation facilitating the process of leader election is Zookeeper.
- If the active services goes down for some reason, another service rises to do its work.
- Acts as a service providing reliable distributed coordination.
- It is highly concurrent, very fast and suitable mainly for read-heavy access patterns.
- Reads can be done against any node of a Zookeeper cluster while writes a quorum-based.
- To reach a quorum, Zookeeper utilizes an atomic broadcast protocol.

Leader Election in Zookeeper

- All participants of the election process create an ephemeral-sequential node on the same election path.
- The node with the smallest sequence number is the leader.
- Each "follower" node listens to the node with the next lower sequence number to prevent a herding effect when the leader goes away.
- In effect this creates a linked list of nodes.
- When a node's local leader dies it goes to election either find a smaller node or becoming the leader if it has the lowest sequence number.

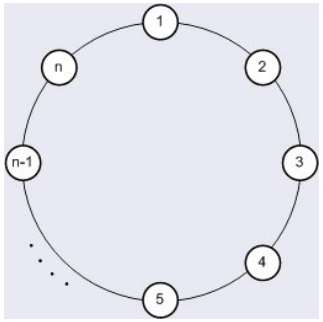
Leader Election in Zookeeper



Ring Networks

- We assume that the network graph G is a ring consisting of n processes.
- Numbered $1 \ldots n$ in the clockwise direction.
- We often count $\mod n$, allowing 0 to be another name for process n, $n+1$ another name for process 1, ...
- The processes associated with the nodes of G do not know their indices, nor those of their neighbors.

Processes in a Ring



- We assume that message-generation and transition functions are defined in terms of local, relative names of their neighbors.

Problem Definition

An algorithm solves the problem of leader election if it meets the following specifications:

1. All halting states are split in two sub sets:
   - 1.1 all states that indicate the process as being "elected",
   - 1.2 all states that indicate the process as being "not-elected".
2. When a process reaches a halting state, the state-transition function only allows it to transit to states of the same subset.
3. In every execution of the algorithm
   - one and only one process is "elected",
   - all other processes are in "not-elected" state.

Variations of the problem

There are several variations of the problem:

- The ring can be either unidirectional or bidirectional.
- The number n of nodes may be either known or unknown to the processes.
- Processes may be identical or can be distinguished by each starting with a unique identifier (UID).
- It might be required that all not-elected processes eventually output the value "non-leader".
- It might be required that all non-elected processes eventually output the UID of the leader.
- We might wish to elect k leaders.
- ...

Anonymous Leader Election

- Computing a leader is a most simple form of symmetry breaking.
- Algorithms based on leaders do generally not exhibit a high degree of parallelism — often suffer from poor time complexity.
- Sometimes it is useful to have a unique process (leader) to make critical decisions in an easy (though non-distributed/centralized) way.
- A first easy observation is that if all the processes are identical, then this problem cannot be solved at all in the given model.

Anonymous Network

A system is anonymous if nodes do not have unique identifiers.

Anonymous Leader Election in Symmetric Networks

If a leader can be elected in an anonymous network depends on the network type:

- In symmetric networks leader election is impossible.
  e.g., ring, complete graph, complete bipartite graph, ...
- In asymmetric networks leader election is possible.
  e.g., star, single node with highest degree, ...

Theorem (2.1)

Let A be a system of n processes, $n > 1$, arranged in a bidirectional ring. If all the processes in A are identical, then A does not solve the leader-election problem.

The idea is that in a ring, symmetry can always be maintained.

Lemma (2.2)

After round k of any deterministic algorithm on an anonymous ring, each node is in the same state s_k.

Proof: Proof by induction:

- All nodes start in the same state.
- A round in a synchronous algorithm consists of the three steps sending, receiving, local computation.
- All nodes send the same message(s), receive the same message(s), do the same local computation,
- Therefore end up in the same state.

■

Symmetry cannot be broken

Proof: Let's suppose that such a system A exists.

- If one node ever decides to become a leader (or a non-leader), then every other node does so as well (due to Lemma 2.2).
- Then all processes become leaders which violates the specifications of the problem.
- If all processes decides to become a non-leader, then the algorithm violates the specifications of the problem.
- In any case, we cannot end up with one and only one leader.

■

Overcoming the Impossibility Result

- Theorem 2.1 implies that the only way to solve the leader-election problem is to break the symmetry somehow.
- A reasonable assumption is to allow processes to be identical except for a UID.
- This is usually done in practice.
- Another possibility is to allow processes to differentiate the execution by using some random factor.

The LCR Algorithm

Algorithm LCR (informal)

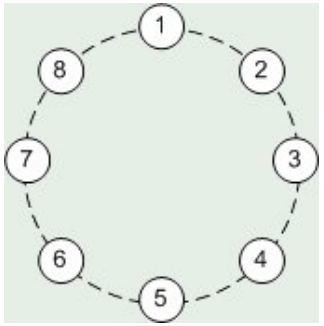
Each process sends its identifier around the ring. When a process receives an incoming identifier, it compares that identifier to its own. If the incoming identifier is greater than its own, it keeps passing the identifier; if it is less that its own, it discards the incoming identifier; if it is equal to its own, the process declares itself the leader.

- Decentralized, Uniform algorithm.
- Uses only unidirectional communication.
- Uses only comparison operations on the UIDs.
- Only the leader performs an output.

Example Execution

- Let's assume a synchronous ring of $n = 8$ processes.
  - Processes are indexed from 1 to 8 clockwise.

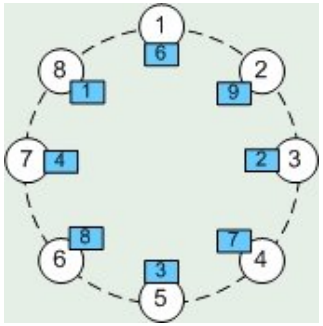
Synchronous Ring



Example Execution

- Let's assume a synchronous ring of $n = 8$ processes.
  - Processes are indexed from 1 to 8 clockwise.
- All processes have UIDs
  - Do now know the UIDs of the other processes.

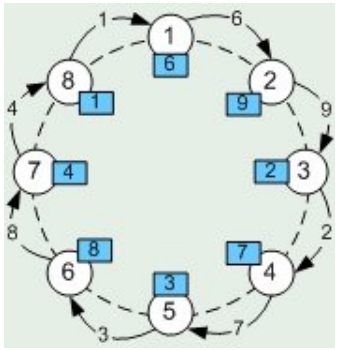
Synchronous Ring



Example Execution

- Let's assume a synchronous ring of $n = 8$ processes.
  - Processes are indexed from 1 to 8 clockwise.
- All processes have UIDs
  - Do now know the UIDs of the other processes.
- First round

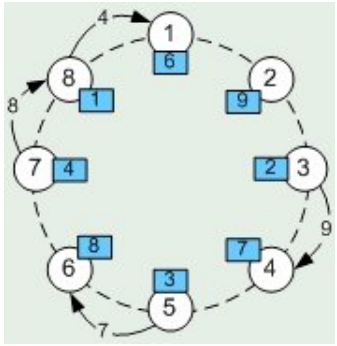
Synchronous Ring



Example Execution

- Let's assume a synchronous ring of $n = 8$ processes.
  - Processes are indexed from 1 to 8 clockwise.
- All processes have UIDs
  - Do now know the UIDs of the other processes.
- First round
- Second round

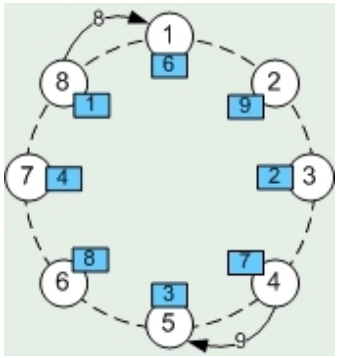
Synchronous Ring



Example Execution

- Let's assume a synchronous ring of $n = 8$ processes.
  - Processes are indexed from 1 to 8 clockwise.
- All processes have UIDs
  - Do now know the UIDs of the other processes.
- First round
- Second round
- Next rounds

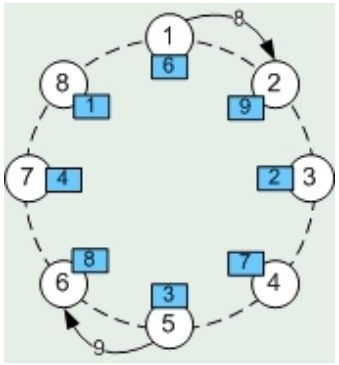
Synchronous Ring



Example Execution

- Let's assume a synchronous ring of $n = 8$ processes.
  - Processes are indexed from 1 to 8 clockwise.
- All processes have UIDs
  - Do now know the UIDs of the other processes.
- First round
- Second round
- Next rounds

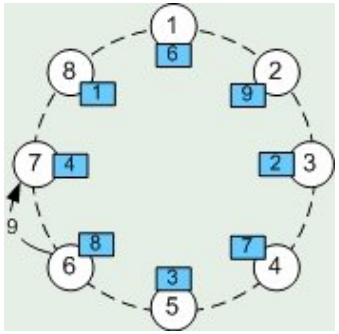
Synchronous Ring



Example Execution

- Let's assume a synchronous ring of $n = 8$ processes.
  - Processes are indexed from 1 to 8 clockwise.
- All processes have UIDs
  - Do now know the UIDs of the other processes.
- First round
- Second round
- Next rounds

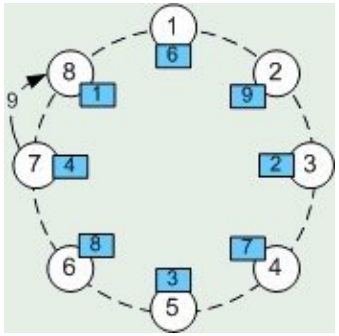
Synchronous Ring



Example Execution

- Let's assume a synchronous ring of $n = 8$ processes.
  - Processes are indexed from 1 to 8 clockwise.
- All processes have UIDs
  - Do now know the UIDs of the other processes.
- First round
- Second round
- Next rounds

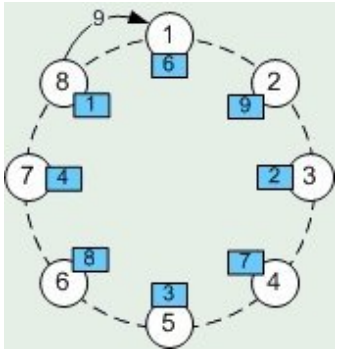
Synchronous Ring



Example Execution

- Let's assume a synchronous ring of $n = 8$ processes.
  - Processes are indexed from 1 to 8 clockwise.
- All processes have UIDs
  - Do now know the UIDs of the other processes.
- First round
- Second round
- Next rounds

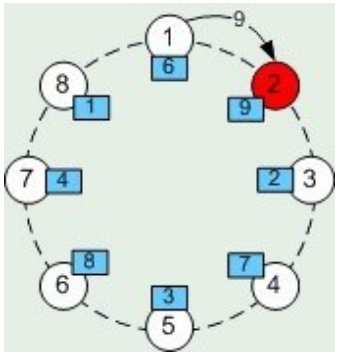
Synchronous Ring



Example Execution

- Let's assume a synchronous ring of $n = 8$ processes.
  - Processes are indexed from 1 to 8 clockwise.
- All processes have UIDs
  - Do now know the UIDs of the other processes.
- First round
- Second round
- Next rounds
- Leader election – process 2

Synchronous Ring



Algorithm Properties

Let i_{max} denote the index of the process with the maximum UID, and u_{max} its UID.

- Process i_{max} is elected leader at the end of round n.
- No other processes apart from i_{max} ends up in "elected" state.
- The time complexity is $\mathcal{O}(n)$
- The message complexity varies...
  - $\mathcal{O}(n^2)$ — worst case,
  - $\mathcal{O}(n)$ — best case,
  - $\mathcal{O}(n \log n)$ — average case.

Proof of Correctness (1)

We denote by $send_i$ the message sent by process i.

Lemma (2.3)

Process i_{max} outputs leader by the end of round n.

Proof: After r rounds, $send_{i_{max}+r}$ contains the UID of i_{max}.

- When $r = 0$ this holds since $send_{i_{max}} = u_{max}$.
- By induction, at the end of round r, process $i_{max} + r$ receives u_{max} and transmits it to the next process, i.e., $send_{i_{max}+r} = i_{max}$.
- Thus, at round n process $i_{max} - 1$ will send to i_{max} the UID u_{max} thus i_{max} will change state to elected.

■

Proof of Correctness (2)

Lemma (2.4)

No process other than i_{max} ever outputs the value leader.

Proof: For any process i at any round r all processes $i_{max}, i_{max+1}, \ldots, i-1, i$ will not transmit the UID of i.
Thus i will never receive its UID – therefore, it will not enter state elected.

■

Theorem (2.5)

LCR solves the leader-election problem.

Message Complexity — worst case

- Processes UIDs are assigned in a descending order.
- Therefore, the UID of i_1 will travel 1 round,
  the UID of i_2 will travel 2 rounds,
  ...
  the UID of i_{max} will travel n rounds
- Thus the total messages exchanged are:

$$1 + 2 + 3 + \ldots + n = \sum_{1}^{n} i = \frac{n(n+1)}{2}$$

- Worst case message complexity: $\mathcal{O}(n^2)$

Message Complexity — best case

- Processes UIDs are assigned in ascending order.
- Therefore, the UID of $i_1 \ldots i_{max} - 1$ will travel 1 round, and the UID of i_{max} will travel n rounds
- Thus the total messages exchanged are:

$$(n-1) + n = 2n - 1$$

- Best case message complexity: $\mathcal{O}(n)$

Message Complexity — average case

- Processes UIDs are assigned uniformly randomly from UID space $1 \ldots n$.
- We observe that
  - The probability of process with index i receiving UID u is the same for all processes and indexes.
  - The process receiving UID u has $u - 1$ processes with smaller UID and $n - u$ processes with larger UID.
- As noted in the previous cases
  - A message sent by process with UID 1 will travel at most 1 round.
  - A message sent by process with UID n will travel at most n rounds.

Message Complexity — average case

- Let $\mathbb{P}_u^i = \frac{u-i}{n-1}$ be the probability process at distance i from process u to have smaller UID.
- Let $\mathbb{P}_u^{*i} = \frac{n-u}{n-i}$ be the probability process at distance i from process u to have bigger UID.
- Let $\mathbb{P}_{u,l}$ be the probability the message of process u to travel l distance
  - Then all processes at distance $l - 1$ have smaller UID from u and the process at distance l has bigger UID.
  - Thus $\mathbb{P}_{u,l}$ is the probability that process u will have the bigger UID from all other processes at distance at most $l - 1$ (clockwise distance) and has smaller UID from the process at distance l.

Message Complexity — average case

Thus,

$$\begin{aligned}
\mathbb{P}_{u,l} &= \mathbb{P}_u^1 \cdot \mathbb{P}_u^2 \cdot \ldots \cdot \mathbb{P}_u^{l-1} \cdot \mathbb{P}_u^{*l} \\
&= \frac{u-1}{n-1} \cdot \frac{u-2}{n-2} \cdot \ldots \cdot \frac{u-1+1}{n-1+1} \cdot \frac{n-u}{n-l} \\
&= \frac{\frac{(u-1)!}{(l-1)!(u-l)!}}{\frac{(n-1)!}{(l-1)!(n-l)!}} \cdot \frac{n-u}{n-l} \\
&= \frac{\binom{u-1}{l-1}}{\binom{n-1}{l-1}} \cdot \frac{n-u}{n-l}
\end{aligned}$$

Message Complexity — average case

Therefore, the expected path length for the messages of u will be

$$\mathbb{E}_u = \sum_{l=1}^{n-1} l \cdot \mathbb{P}_{u,l} \qquad \text{for } u = 1, 2, \ldots, n-1$$

Message Complexity — average case

The total expected number of messages exchanged during the execution of the algorithm is,

$$\begin{aligned}
\mathbb{E} &= n + \sum_{u=1}^{n-1} \sum_{l=1}^{n-1} l \cdot \mathbb{P}_{u,l} \\
&= \ldots \\
&= n + \sum_{l=1}^{n-1} \frac{n}{l+1} \\
&= n\left(1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}\right) \\
&= n\,(C + \ln n) \\
&\Rightarrow \mathcal{O}\,(n \log n)
\end{aligned}$$

Seminal Papers

1. Dana Angluin: Local and Global Properties in Networks of Processors (Extended Abstract). STOC 1980: 82-93
2. Gérard Le Lann: Distributed Systems - Towards a Formal Approach. IFIP Congress 1977: 155-160
3. Ernest J. H. Chang, Rosemary Roberts: An Improved Algorithm for Decentralized Extrema-Finding in Circular Configurations of Processes. Commun. ACM 22(5): 281-283 (1979)

In a monastery monks have to wear a black or white hat.

- They do not know the color of their hat.
- They are not allowed to see the hat they are wearing – or take it out.
- Monks are not allowed to communicate via any mean.
- No Mirrors are allowed in the monastery.
- Only the abbot of the monastery is allowed to talk.

One day the abbot calls all monks to assembly.
— He asks them to perform a set of simple tasks.
— The monks stare each other, and then (without any communication) correctly from two groups based on the color of their hat.

In a monastery monks have to wear a black or white hat.

- They do not know the color of their hat.
- They are not allowed to see the hat they are wearing – or take it out.
- Monks are not allowed to communicate via any mean.
- No Mirrors are allowed in the monastery.
- Only the abbot of the monastery is allowed to talk.

Open Problem # 1

What distributed algorithm did the monks execute?

- Assume a synchronous unidirectional ring of n processes.
- Each process has a unique identity.
- Processes are not aware of the total number of processes.
- Processes are capable of evaluating if two UIDs are equal or not.
- No other operand is implemented.

Open Problem # 2

Does the problem variation has a solution or not?