Synchronous and asynchronous model: Leader election in a ring

IMT Atlantique, Département d’Informatique, Brest
Equipe Math Et Net

Synchronous Distributed Algorithm

Definition: In a synchronous distributed algorithm, nodes operate in **synchronous rounds** (they have access to a **global clock**). They execute:

(1) local computation

(2) send messages to other processes

(3) receive/read messages from other processes

Synchronous Distributed Algorithm

Definition: In a synchronous distributed algorithm, nodes operate in **synchronous rounds** (they have access to a **global clock**). They execute:

(1) local computation

(2) send messages to other processes

(3) receive/read messages from other processes

```go
package main

import (
  "fmt"
  "time"
)

func say(s string) {
  for i := 0; i < 1; i++ {
    fmt.Println(s)
  }
}

func main() {
  go say("world")
  fmt.Println("Hello")
  time.Sleep(1000 * time.Millisecond)


}
```

Synchronous Distributed Algorithm

Definition: In a synchronous distributed algorithm, nodes operate in **synchronous rounds** (they have access to a **global clock**). They execute:

(1) local computation

(2) send messages to other processes

(3) receive/read messages from other processes

Synchronous distributed algorithm ?

```go
package main

import (
  "fmt"
  "time"
)

func say(s string) {
  for i := 0; i < 1; i++ {
    fmt.Println(s)
  }
}

func main() {
  go say("world")
  fmt.Println("Hello")
  time.Sleep(1000 * time.Millisecond)


}
```

Synchronous Distributed Algorithm

Definition: In a synchronous distributed algorithm, nodes operate in **synchronous rounds** (they have access to a **global clock**). They execute:

(1) local computation

(2) send messages to other processes

(3) receive/read messages from other processes

Synchronous distributed algorithm ?

```go
package main

import (
  "fmt"
  "time"
)

func say(s string) {
  for i := 0; i < 1; i++ {
    fmt.Println(s)
  }
}

func main() {
  go say("world")
  fmt.Println("Hello")
  time.Sleep(1000 * time.Millisecond)


}
```

Definition: For synchronous distributed algorithm, the **time complexity** is **the number of rounds** until the algorithm **terminates.**

Asynchronous Distributed Algorithm

Definition: In the asynchronous model, algorithms are:

- Event driven;
- Processors cannot access a global clock;
- Communication between two processors will occur in finite but unbounded time.

Asynchronous Distributed Algorithm

Definition: In the asynchronous model, algorithms are:

- Event driven;
- Processors cannot access a global clock;
- Communication between two processors will occur in finite but unbounded time.

```go
package main

import (
  "fmt"
  "time"
  )

type Ball struct{ hits int }

func main() {
    table := make(chan *Ball)
    go player1("Alice", table)
    go player2("Bob", table)

    table <- new(Ball) // game on; toss the ball
    time.Sleep(1 * time.Second)
    <-table // game over; grab the ball
}

func player1(name string, table chan *Ball) {
    for {
        ball := <-table
        ball.hits++
        fmt.Println(name, ball.hits)
        time.Sleep(100 * time.Millisecond)
        table <- ball
    }
}
func player2(name string, table chan *Ball) {
    for {
        ball := <-table
        ball.hits++
        fmt.Println(name, ball.hits)
        time.Sleep(200 * time.Millisecond)
        table <- ball
    }
}
```

Asynchronous Distributed Algorithm

Definition: In the asynchronous model, algorithms are:

- Event driven;
- Processors cannot access a global clock;
- Communication between two processors will; occur in finite but unbounded time.

Asynchronous distributed algorithm ?

```go
package main

import (
  "fmt"
  "time"
  )

type Ball struct{ hits int }

func main() {
    table := make(chan *Ball)
    go player1("Alice", table)
    go player2("Bob", table)

    table <- new(Ball) // game on; toss the ball
    time.Sleep(1 * time.Second)
    <-table // game over; grab the ball
}

func player1(name string, table chan *Ball) {
    for {
        ball := <-table
        ball.hits++
        fmt.Println(name, ball.hits)
        time.Sleep(100 * time.Millisecond)
        table <- ball
    }
}
func player2(name string, table chan *Ball) {
    for {
        ball := <-table
        ball.hits++
        fmt.Println(name, ball.hits)
        time.Sleep(200 * time.Millisecond)
        table <- ball
    }
}
```

Asynchronous Distributed Algorithm

Definition: In the asynchronous model, algorithms are:

- Event driven;
- Processors cannot access a global clock;
- Communication between two processors will; occur in finite but unbounded time.

Asynchronous distributed algorithm ?

```go
package main

import (
  "fmt"
  "time"
  )

type Ball struct{ hits int }

func main() {
    table := make(chan *Ball)
    go player1("Alice", table)
    go player2("Bob", table)

    table <- new(Ball) // game on; toss the ball
    time.Sleep(1 * time.Second)
    <-table // game over; grab the ball
}

func player1(name string, table chan *Ball) {
    for {
        ball := <-table
        ball.hits++
        fmt.Println(name, ball.hits)
        time.Sleep(100 * time.Millisecond)
        table <- ball
    }
}
func player2(name string, table chan *Ball) {
    for {
        ball := <-table
        ball.hits++
        fmt.Println(name, ball.hits)
        time.Sleep(200 * time.Millisecond)
        table <- ball
    }
}
```

The notion of rounds does not exist here.

Definition: The time complexity is **the number of time units** from the start of the execution to its completion in the **worst case**.

Asynchronous Distributed Algorithm

Definition: In the asynchronous model, algorithms are:

- Event driven;
- Processors cannot access a global clock;
- Communication between two processors will; occur in finite but unbounded time.

Asynchronous distributed algorithm ?

```go
package main

import (
  "fmt"
  "time"
  )

type Ball struct{ hits int }

func main() {
    table := make(chan *Ball)
    go player1("Alice", table)
    go player2("Bob", table)

    table <- new(Ball) // game on; toss the ball
    time.Sleep(1 * time.Second)
    <-table // game over; grab the ball
}

func player1(name string, table chan *Ball) {
    for {
        ball := <-table
        ball.hits++
        fmt.Println(name, ball.hits)
        time.Sleep(100 * time.Millisecond)
        table <- ball
    }
}
func player2(name string, table chan *Ball) {
    for {
        ball := <-table
        ball.hits++
        fmt.Println(name, ball.hits)
        time.Sleep(200 * time.Millisecond)
        table <- ball
    }
}
```

The notion of rounds does not exist here.

Definition: The time complexity is **the number of time units** from the start of the execution to its completion in the **worst case**.

Leader Election

Sometime, it is needed to elect a process as a **leader** (to make critical decisions, to commit the final decision).

What are the problems with an algorithm based on a leader?

Leader Election

Sometime, it is needed to elect a process as a **leader** (to make critical decisions, to commit the final decision).

What are the problems with an algorithm based on a leader?

Breaking symmetry, not robust to failure

Problem definition (Leader Election): Each node eventually decides whether it is a leader or not, subject to the constraint that there is exactly one leader.

Leader Election

Sometime, it is needed to elect a process as a **leader** (to make critical decisions, to commit the final decision).

What are the problems with an algorithm based on a leader?

Breaking symmetry, not robust to failure

Problem definition (Leader Election): Each node eventually decides whether it is a leader or not, subject to the constraint that there is exactly one leader.

Leader Election

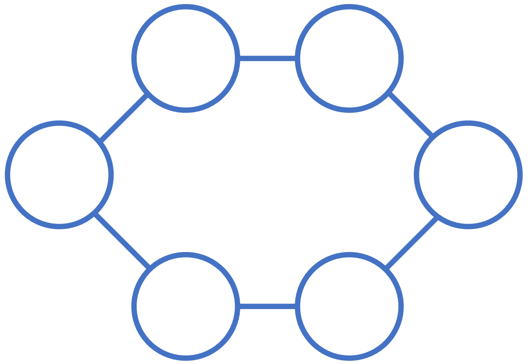
Sometime, it is needed to elect a process as a **leader** (to make critical decisions, to commit the final decision).

What are the problems with an algorithm based on a leader?

Breaking symmetry, not robust to failure

Problem definition (Leader Election): Each node eventually decides whether it is a leader or not, subject to the constraint that there is exactly one leader.

We will study the Leader Election on ring topology



Formally

Each processor has a set of elected states (« *I'm a leader* ») and a set of non-elected states (« *I'm a follower* »). Once a process enters in an elected/non-elected state, it cannot exit that state

For every admissible execution

Liveness property: ?

Safety property: ?

Formally

Each processor has a set of elected states (« *I'm a leader* ») and a set of non-elected states (« *I'm a follower* »). Once a process enters in an elected/non-elected state, it cannot exit that state

For every admissible execution

Liveness property: At some point, every processor is in an elected state or in a non-elected state

Safety property: ?

Formally

Each processor has a set of elected states (« *I'm a leader* ») and a set of non-elected states (« *I'm a follower* »). Once a process enters in an elected/non-elected state, it cannot exit that state

For every admissible execution

Liveness property: At some point, every processor is in an elected state or in a non-elected state

Safety property: one processor enters an elected state

A first impossibility theorem

Assumptions

A first impossibility theorem

Assumptions

Anonymous system: nodes do not have a unique ID.

Uniform algorithm: Number of nodes is not known to the algorithm or the nodes.

A first impossibility theorem

Assumptions

Anonymous system: nodes do not have a unique ID.

Uniform algorithm: Number of nodes is not known to the algorithm or the nodes.

Lemma: If all the processes start in the same state, then for every $k > 0$, for every deterministic algorithm on an anonymous ring, each node is in the same state at each step k.

Proof:

A first impossibility theorem

Assumptions

Anonymous system: nodes do not have a unique ID.

Uniform algorithm: Number of nodes is not known to the algorithm or the nodes.

Lemma: If all the processes start in the same state, then for every $k > 0$, for every deterministic algorithm on an anonymous ring, each node is in the same state at each step k.

Proof: Let us study the synchronous case. Let x_i^n being the state of processor i at time n. Let y_i^n being the message sent to processor $i+1$ at instant n. We assume that the algorithm has the following update rule:

$$x_n^i = f^n(x_{n-1}^i, y_n^{i-1 \ mod \ I-1})$$

$$y_n^i = g^n(x_n^i)$$

With $x_i^0 = x_0$ for all i. Note that by simply using a induction argument you will be able to finish the proof.

A first impossibility theorem

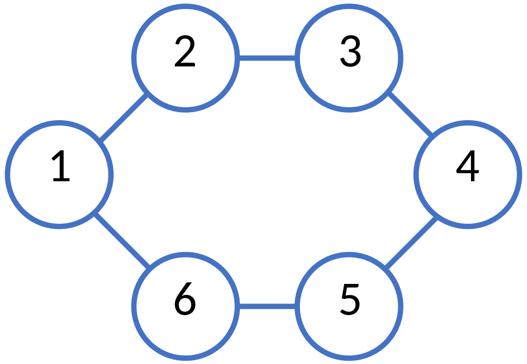
Theorem: If all the processes are the same and start in the same state, deterministic leader election in anonymous ring is impossible

Proof: Use the previous lemma. Note that if one node decides to be a leader, then every node will do the same. Moreover, note that it is true for uniform and non-uniform algorithm.

Our assumptions are too strong. We need to relax one of them

Non-anonymous system: nodes do have an unique ID.

Uniform algorithm: Number of nodes is not known to the algorithm or the nodes.



Asynchronous Ring

Algorithm (Clockwise)

For each node i

$s_i := i$

Upon i receive no message: send s_i to $j' = i + 1\ mod\ I$

Upon i receive a message x_j from $j := i - 1\ mod\ I$

(Case 1) $x_j > s_i$:

- $s_i = x_j$, « I'm not the leader », Send s_i to $j' = i + 1\ mod\ I.$

(Case 2) $x_j < s_i$:

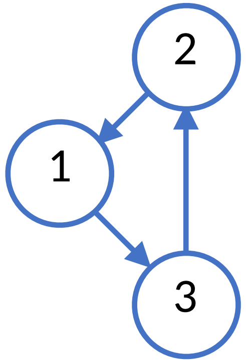
- « I don't know if I'm the leader », Discard the message x_j

(Case 3) $x_j = s_i$:

- « I'm the leader» Send $< s_i, terminate >$ to $j' = i + 1\ mod\ I$, terminate.

Upon receiving $< x_j, terminate >$: $s_i = x_j$, **send** $< s_i, terminate >$, terminate.

Example



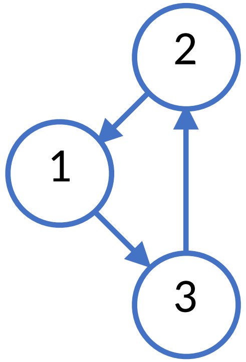
Execution 1: [1,2,3], $send_{13}(1)$, [1,2,3], $send_{32}(3)$, [1,3,3], $send_{21}(3)$, [3,3,3],$send_{13}(3)$, [3,3,<I'm the leader,Terminate>], etc…

Associated Trace: $send_{13}(1)$ $send_{32}(3)$ $send_{21}(3)$ $send_{13}(3)$, etc…

Execution 2: [1,2,3], $send_{13}(1)$, [1,2,3], $send_{21}(2)$, [2,2,3], $send_{32}(3)$, [2,3,3], $send_{21}(2)$,[2,3,3],$send_{21}(3)$,[3,3,3],$send_{13}(3)$, [3,3,<I'm the leader,Terminate>], etc…

Associated Trace: $send_{13}(1)$, $send_{21}(2)$, $send_{32}(3)$, $send_{21}(2)$,$send_{21}(3)$, $send_{13}(3)$, etc…

Example



Execution Fragment

Execution 1: [1,2,3], $send_{13}(1)$, $\boxed{[1,2,3],\ send_{32}(3),\ [1,3,3],\ send_{21}(3),}$ [3,3,3],$send_{13}(3)$, [3,3,<I'm the leader,Terminate>], etc…

Associated Trace: $send_{13}(1)$ $send_{32}(3)$ $send_{21}(3)$ $send_{13}(3)$, etc…

Execution 2: [1,2,3], $send_{13}(1)$, [1,2,3], $send_{21}(2)$, [2,2,3], $send_{32}(3)$, [2,3,3], $send_{21}(2)$,[2,3,3],$send_{21}(3)$,[3,3,3],$send_{13}(3)$, [3,3,<I'm the leader,Terminate>], etc…

Associated Trace: $send_{13}(1)$, $send_{21}(2)$, $send_{32}(3)$, $send_{21}(2)$,$send_{21}(3)$, $send_{13}(3)$, etc…

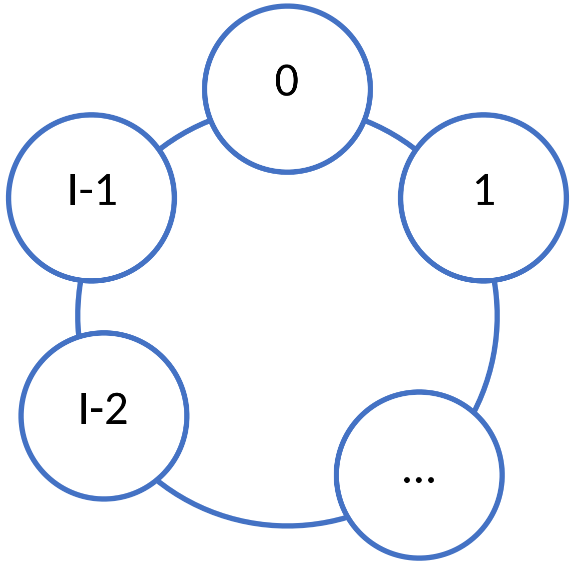
Complexity analysis

Definition: The **message complexity** of an algorithm is the number of messages exchanged until the algorithm completes his task.

Theorem: The algorithm is solving the leader election problem (liveness and safety properties are satisfied). The message complexity is $O(I^2)$.

Proof: First point can be proven by noticing that the process with the higher index will reject all the messages except the ones coming with his Id.

Exercice (Message complexity):



Synchronous algorithm

Assumptions:

- Ids are positive integers (for instance, 10, 20, 21)
- I is known to all processors
- We assume that every node starts at the same time.
- The node with the minimum identifier becomes the leader.

Algorithm (Timeslice algorithm)

Inputs: Each phase is composed of I time steps.

If a process i exists with UID v_i, then if round (v_i-1)I +1 is reached without i have previously received a non-null message, the i elects itself leader and circulates a token with its UID around the ring.

Exercice 1: Specify the algorithm using the state machine notation.

Exercice 2: What is the time and communication complexity?

Acknowledgements

This course is mainly based on: http://ac.informatik.uni-freiburg.de/teaching/ss_15/netalg/LectureNotes/chapter3.pdf

To know more about leader elections problems (lower bound message complexity, leader election in a general network)

https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-852j-distributed-algorithms-fall-2009/lecture-notes/MIT6_852JF09_lec02.pdf

https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-852j-distributed-algorithms-fall-2009/lecture-notes/MIT6_852JF09_lec08.pdf