leader election &
mutual exclusion
- part II
Homework

How to elect a leader on a general graph (Q2.2)
- unique identifiers
- synchronous system

Let every node keep a set $I$
- initially $I = \{ CEiJ.id \}$

In every round
- send $I$ to all neighbours
- receive $I'$ from neighbours, and merge that with $I$
- continue this until $I$ never grows
- decide if $CEiJ.id$ is maximum in $I$

$\Rightarrow 0 \div \%$
Lamport’s logical clock algorithm

- every event a at node i is assigned a natural number $C_i(a)$
- the logical clock of an action a (on node i) $C(a) = C_i(a)$
- correctness condition: $a \rightarrow b$ then $C(a) \leq C(b)$

Implementation

- every node i keeps a counter $C_i$
- initially $C_i = 0$
- $C_i$ is incremented whenever a node executes an action
- $C_i(a)$ is the value of $C_i$ just before executing a

- suppose a is a send event (with $C_i(a) = x$)
- tag the send message m with $T_m = C_i(\text{send})$
- when receiving a message m with a action b on node j, set $c_j \leftarrow \max (c_j, T_m + 1)$; $C_i(b) = c_j$
Q: How to do this with shared variables

A: Every shared variable $v$ is tagged with $T_v$

- Writing: setting $T_v$ to the clock of the write action
- Reading: setting the clock $c_j$ of the reader $j$ to
  $\max (c_j, T_v + 1)$; $c_j(\text{read}) = c_j$

Clock values also contain the node identifier

$\langle c, i \rangle \triangleq$ lexicographically ordered

$\Rightarrow a \Rightarrow b \lor b = c \Rightarrow a$

$a \neq a$
Petersen leader election

The protocol for a node $i$:

$$C[i].active = \text{true}$$

$$C[i].leader = \text{false}$$

$\textbf{while true}$

$\textbf{do}$

$\textbf{if} \ C[i].active \land \neg C[i].leader$

$\textbf{then}$

send left $C[i].id \leftarrow$

send right $C[i].id \leftarrow$

receive left $leftid$

receive right $rightid$

$\textbf{if} \ C[i].id \leq leftid \land C[i].id \leq rightid$

$\textbf{then}$

$C[i].leader \leftarrow \text{true}$

$\textbf{else}$

$\textbf{if} \ C[i].id < leftid \lor C[i].id < rightid$

$\textbf{then}$

$C[i].active \leftarrow \text{false}$

$\textbf{else}$

(\* node passive or a leader \*)

receive left $leftid$; send right $leftid$

receive right $rightid$; send left $rightid$ 3 relay
How efficient is the Peterson leader election protocol?

\[ \alpha, \beta, \gamma \]

the largest identifier wins

the number of active nodes is halved every time
Can we do this unidirectionally?

- Clockwise ring

Active

Simulated the bidirectional ring on the unidirectional ring

Passive

by moving counterclockwise neighbours 'virtually' one step clockwise
Mutual exclusion — Lamport’s Bakery algorithm

Properties
- mutual exclusion: only one process in the CS.
- (no deadlock) progress
- (fair) no starvation

while true
    → enter()
        (at critical section *)
    → exit()
        (at remainder section *)
Lamport's bakery algorithm  \( I = i_1, \ldots, i_n \)

each node \( i \) has two variables: shared, so everybody else can read them (atomic)

- \( C[i].\text{num} : \text{unbounded}! \) (num \( [i] \))
- \( C[i].\text{choosing} : \text{boolean} \) (choosing \( i \))

\[
\begin{aligned}
\text{num} [i] & \leftarrow 0 \\
\text{choosing} [i] & \leftarrow \text{false}
\end{aligned}
\]

while true
\[
\begin{aligned}
\text{choosing} [i] & \leftarrow \text{true} \\
\text{num} [i] & \leftarrow \max \{ j : \text{num} [j] + 1 \\
\text{choosing} [i] & \leftarrow \text{false}
\end{aligned}
\]

for \( j \neq i \)
\[
\begin{aligned}
\text{do while} & \ (\text{choosing} [j] \ \text{do} \ (\neq \text{noop} / \text{wait} \neq)) \\
\text{while} & \ (\text{num} [j] > 0) \ \land \ ((\text{num} [i,j],j) \leq (\text{num}[i],i)) \\
\text{do} & \ (\neq \text{wait} / \text{noop} \neq)
\end{aligned}
\]

\(\text{critical section} \neq \)

\[
\begin{aligned}
\text{exit} & \ [ \text{num} [i] \leftarrow 0 \]
\end{aligned}
\]
Main lemma \[ \text{if } i \text{ is in the C.S., and there is a node } k \text{ s.t. } \text{num}[k] \neq 0 \text{ then } (\text{num}[k], k) > (\text{num}[i], i) \]

- Before entering, node \( i \) first waited until choosing \( [k] = \text{false} \) and after that it waited until either
  1) \( \text{num}[k] = 0 \)
  2) \( (\text{num}[k], k) > (\text{num}[i], i) \)

- In case 1), because we now have \( \text{num}[k] \neq 0 \), node \( k \) must have changed \( \text{num}[k] \) after node \( i \) read it
  \[ \text{read } \text{num}[k] \rightarrow \text{write } \text{num}[k] \]

Node \( i \) sets its current value of \( \text{num}[i] \) before reading choosing \( [k] = \text{false} \) and reading num.

- \( \text{write } \text{num}[i] \rightarrow \text{read choosing } [k] = \text{false} \rightarrow \text{read } \text{num}[k] \)

But node \( k \) sets choosing \( [k] \) to true before writing \( \text{num}[k] \) and reading \( \text{num}[i] \)

- \( \text{write choosing } [k] \text{ true} \rightarrow \text{read } \text{num}[i] \rightarrow \text{write } \text{num}[k] \)

This can only happen if \( \text{read choosing } [k] = \text{false} \rightarrow \text{write or choosing } [k] \) is true
This guarantees that he sees the value of \( n + 1 \) and hence chooses a larger ticket (because of the \(+ 1\) in the ticket formula)