Leader Election

Why Rings?

- Historical reasons
  - original motivation: regenerate lost token in token ring networks
- Illustrates techniques and principles
- Good for lower bounds and impossibility results

The Idea

We study leader election in rings

Outline

- Specification of Leader Election
- YAIR
- Leader Election in Asynchronous Rings
  - An $O(n^2)$ algorithm
  - An $O(n \log(n))$ algorithm
- The Revenge of the Lower Bound
- Leader election is synchronous rings
  - Breaking the $\Omega(n \log(n))$ barrier
The Problem

- Processes can be in one of two final states: **elected** or **non-elected**
- In every execution, exactly one process (the **leader**) is elected
- All other processes are non-elected

Lots of variations...

- Ring can be **unidirectional** or **bidirectional**
- Processes can be **identical** or can somehow be **distinguishable** from each other
- The number \( n \) of processes may or may not be known - if not, **uniform** algorithms
- Communications may be **synchronous** or **asynchronous**

Anonymous Networks

- Processes have no unique IDs (identical automata)
- ...but can distinguish between left and right

Call me Ishmael

- Processes have unique IDs from some large totally ordered set (e.g. \( \mathbb{N}^+ \))
- Operations used to manipulate IDs can be unrestricted or limited (e.g. only comparisons)
Communication:
Synchronous/Asynchronous

- Synchronous
  - In rounds
  - In each round, a process
    1. delivers all pending messages
    2. takes an execution step (possibly sending one or more messages)

- Asynchronous
  - No upper bound on message delivery time
  - No centralized clock
  - No bound on relative speed of processes

An Impossibility Result

Theorem
There is no nonuniform anonymous algorithm for leader election in synchronous rings

Proof
Suppose there exists an anonymous nonuniform algorithm $A$ for $R$ s.t. $|R| > 1$

Lemma For every round $k$ of $A$ in $R$, the states of all the processes at the end of round $k$ are the same

Proof By induction on $k$
If some process enters the leader state, they all do

An $O(n^2)$ Algorithm
Le Lann (’77), Chang & Roberts (’79)

- Asynchronous and Uniform
- Process with highest uid is elected leader - all other uids are swallowed
- Time complexity: $O(n)$
- Message complexity: $O(n^2)$
- Bound is tight:

\[
\begin{align*}
\text{upon receiving no message} & \quad \text{send } uid, \text{ to left (clockwise)} \\
\text{upon receiving } m \text{ from right case } & \quad m.uid > uid, \\
& \quad \text{send } m \text{ to left} \\
& \quad m.uid < uid, \\
& \quad \text{discard } m \\
& \quad m.uid = uid, \\
leader := i \\
& \quad \text{send } \langle \text{terminate, i} \rangle \text{ to left} \\
\text{terminate endcase} & \quad \text{upon receiving } \langle \text{terminate, i} \rangle \text{ from right leader := i} \\
& \quad \text{send } \langle \text{terminate, i} \rangle \text{ to left} \\
& \quad \text{terminate}
\end{align*}
\]
Bidirectional ring

In each phase $k$, $p_i$:
- Sends $uid_i$ token left and right
- Token intended to travel distance $2^k$ and turn back
- Continues outbound only if greater than tokens on path
- Processes always forward inbound token
- $p_i$ leader if it receives own token while going outbound

An $O(n \log n)$ Algorithm
Hirschenberg & Sinclair (1980)

$\forall i$: $uid_i \geq 2^k$

$\forall k$: $|\text{ leadership degree of } p_i| \leq 2k+1$

Phase 0
$O(\log n)$

Phase 1
$O(\log n)$
An $O(n \log n)$ Algorithm
Hirschenberg & Sinclair (1980)

- Bidirectional ring
- In each phase $k$, protocol elects one winner (process with highest uid) for each $k$-neighborhood
  - A $k$-neighborhood includes $2k+1$ processes
- After $O(\log n)$ phases, there is only one winner!

Bounding message complexity

**Lemma** For every $k \geq 1$ the number of processes that are phase $k$ winners are at most $\frac{n}{n+1}$

**Proof** Two winners cannot have fewer than $2^k$ processes between them

Message complexity:

$$4n + \sum_{k=1}^{\lceil \log(n-1) \rceil + 1}$$
Bounding message complexity

Lemma For every \( k \geq 1 \) the number of processes that are phase \( k \) winners are at most \( \frac{n}{2^k+1} \)

Proof Two winners cannot have fewer than \( 2^k \) processes between them

Message complexity:

\[
\begin{align*}
\text{Phase 0:} & \quad 4n + \sum_{k=1}^{[\log(n-1)]+1} 4 \cdot 2^k \\
\text{termination:} & \quad \sum_{k=1}^{[\log(n-1)]+1} \frac{4 \cdot 2^k \cdot n}{2^{k-1} + 1}
\end{align*}
\]
The Revenge of the Lower Bound

- We have seen:
  - a simple \(O(n^2)\) algorithm
  - a more clever \(O(n \log n)\) algorithm

- Facts
  - \(\Omega(n \log n)\) lower bound in asynchronous networks
  - \(\Omega(n \log n)\) lower bounds in synchronous networks when using only comparisons

Breaking through \(\Omega(n \log n)\)

- Synchronous rings
- UID are positive integers, manipulated using arbitrary operations

<table>
<thead>
<tr>
<th>Non Uniform</th>
<th>Uniform</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n) is known to all</td>
<td>(n) is not known</td>
</tr>
<tr>
<td>unidirectional communication</td>
<td>unidirectional communication</td>
</tr>
<tr>
<td>(O(n)) messages!</td>
<td>(O(n)) messages!</td>
</tr>
</tbody>
</table>

What about time complexity?

And now, for something completely different...

RANDOMIZATION

What is it good for?

- In general does not affect
  - impossibility results
  - leader election in anonymous networks
  - worst case bounds
  - consensus in fewer than \(f+1\) rounds

- But it makes a difference when combined with weakening the problem statement
Randomized leader election

- Transition function takes as input
  - a random number
  - from a bounded range
  - under some fixed distribution

- Weaker problem definition for LE:
  - **Safety:** In every global state of every execution, at most one process is in the elected state
  - **Liveness:** At least one process is elected with some non-zero probability

A second look at anonymous rings

**Theorem**

There is a randomized algorithm that, with probability \( c > \frac{1}{e} \) elects a leader in a synchronous ring sending \( O(n^2) \) messages

The “one-shot” algorithm

- Initially
  - \( id_i = \begin{cases} 1 & \text{with probability } 1 - \frac{1}{n} \\ 2 & \text{with probability } \frac{1}{n} \end{cases} \)
  - send \((id_i)\) to left
- upon receiving \((S)\) from right
  - if \(|S| = n\) then
    - if \(id_i\) is unique maximum of \(S\) then
      - elected := true
    - else
      - elected := false
  - else
    - send \((S \cdot id_i)\) to left

- One execution for each element of \( R = \{1, 2\}^n \)
- Algorithm terminates when exactly one process has \( id = 2 \)
- Probability of termination \( c \) :
  \[
  \left( \frac{n}{1} \right) \frac{1}{n} \left( 1 - \frac{1}{n} \right)^{n-1} = \left( 1 - \frac{1}{n} \right)^{n-1}
  \]
  \[
  c > \left( 1 - \frac{1}{n} \right)^n \to \frac{1}{e}
  \]
- Message complexity: \( O(n^2) \)

The iterated algorithm

- If one execution does not terminate with a leader, try again!
- How many times?
  - In the worst case, infinitely many!
  - But in the expected case?
The iterated algorithm

- If one execution does not terminate with a leader, try again!

- How many times?
  - In the worst case, infinitely many!
  - But in the expected case?
    - Expected value of $T$: $E[T] = \sum_{x \in T} x \cdot Pr[T = x]$
    - Probability of success in iteration $i$: $c \cdot (1 - c)^{i-1}$
    - Expected number of iterations:
      $\sum_{i=0}^{\infty} i \cdot c \cdot (1 - c)^{i-1} = -c \cdot \frac{d}{dc} \sum_{i=0}^{\infty} (1 - c)^i = -c \cdot \frac{d}{dc} \frac{1}{1 - (1 - c)} = 1/c < e$

Summary

- No deterministic solution for anonymous rings
- No solution for uniform anonymous rings (even when using randomization)
- Protocols with $O(n^2)$ and $O(n \log n)$ messages for uniform rings
- $\Omega(n \log n)$ lower bound on message complexity for practical protocols
- $O(n)$ message complexity for uniform synchronous rings
**Clock Synchronization**

What is the time?

Hard truth: clocks drift apart

**Clock Drift**

- **Bound on drift:** $\rho$
  
  $$(1 - \rho)(t - t') \leq H(t) - H(t') \leq (1 + \rho)(t - t')$$

- $\rho$ is typically small ($10^{-6}$)
  
  $\rho^2 \approx 0$
  
  $\frac{1}{1 - \rho} = 1 + \rho$
  
  $\frac{1}{1 + \rho} = 1 - \rho$

- **External Clock Synchronization:** keeps clock within some maximum deviation from an external time source.
  - Exchange of info about timing events of different systems
  - Can take actions at real-time deadlines

- **Internal Clock Synchronization:** keeps clocks within some maximum deviation from each other.
  - Can measure duration of distributed activities that start on one process and terminate on another
  - Can totally order events that occur in a distributed system

**External vs Internal Synchronization**

12:00

13:00

Wait... what???

There you go...
Probabilistic Clock Synchronization (Cristian)

- Master-Slave architecture
- Master can be connected to external time source
- Slaves read master’s clock and adjust their own

How accurately can a slave read the master’s clock?

The Idea

- Clock accuracy depends on message roundtrip time
- If roundtrip is small, master and slave cannot have drifted by much!
- No upper bound on message delivery, so no certainty of accurate enough reading...
- ... but very accurate reading can be achieved by repeated attempts

Setup and assumptions

Goal: Synchronize the slave’s clock with the master

Assume that minimum delay is known
Assume that clock drifts are known ($\rho$ for both)

The protocol

Question: what is $Q(x)$?
Ideal scenario

Assume no clock drift

Perfect synchronization!

Problem #1: message delay

Problem #2: slave drift

$2d(1 - \rho) \leq 2D \leq 2d(1 + \rho)$

Problem #3: master drift

During the master’s clock drifts

Even if you know $\beta$, there is still some uncertainty!
Cristian’s algorithm

Naive estimation: $Q(x) = T + (\min + \beta)$
(take master’s drift into account)

$Q(x) \in [T + (\min + \beta)(1 - \rho), T + (\min + \beta)(1 + \rho)]$

$0 \leq \beta \leq 2d - 2\min$ (take delay into account)

$Q(x) \in [T + (\min + 0)(1 - \rho), T + (\min + 2d - 2\min)(1 + \rho)]$

$= [T + (\min)(1 - \rho), T + (2d - \min)(1 + \rho)]$

$2d \leq 2D(1 + \rho)$ (take slave’s drift into account)

$Q(x) \in [T + (\min)(1 - \rho), T + 2D(1 + 2\rho) - \min(1 + \rho)]$

= $[T + (\min)(1 - \rho), T + 2D(1 + 2\rho) - \min(1 + \rho)]$

Slave’s estimation and precision

Slave’s best guess: $Q(x) = T + D(1 + 2\rho) - \min \cdot \rho$

Maximum error: $e = D(1 + 2\rho) - \min$

You can keep trying, until you achieve the required precision

Adjusting the clock

After synchronizing:

- If slave simply sets $P(x) = Q(x)$, it could create time discontinuities.
**Adjusting the clock**

Logical clock: \( C(t) = H(t) + A(t) \)

- **Hardware clock**
- **Adjustment function**

Use linear adjustment function: \( A(t) = mH(t) + N \)

\[
H(t) = C(t) - A(t) = H(t) - (mH(t) + N) = (1 - m)H(t) + N
\]

**Network Time Protocol**

- The oldest distributed protocol still running on the Internet
- Hierarchical architecture
- Latency-tolerant, jitter-tolerant, fault-tolerant... very tolerant!

**Hierarchical structure**

Each level is called a "stratum"

- **Stratum 0**: atomic clocks
- **Stratum 1**: time servers with direct connections to stratum 0
- **Stratum 2**: Use stratum 1 as time sources and work as server to stratum 3
- etc...

Accuracy is loosely coupled with stratum level
Very tolerant. How?

- Tolerance to jitter, latency, faults: redundancy
- Each machine sends NTP requests to many other servers on the same or the previous stratum
- The synchronization protocol between two machines is similar to Cristian’s algorithm
- For each response, we generate a tuple \( <T, \delta> \) which defines an interval \([T-\delta, T+\delta]\)
- How to combine those intervals?

Marzullo’s algorithm

- Given M source intervals, find the largest interval that is contained in the largest number of source intervals

Marzullo’s algorithm

- Given M source intervals, find the largest interval that is contained in the largest number of source intervals

The intuition

- Visit the endpoints left-to-right
- Count how many source intervals are active at each time
- Increase count at starting points, decrease at ending points
**Preprocessing**

- For each source interval $[T_1, T_2]$, create 2 tuples of the form `<time, type>`:
  - `<T_1, -1>` (start of interval)
  - `<T_2, +1>` (end of interval)
- Sort all tuples according to time

Example:
- Source intervals: $[8,12], [11,13], [14,15]$
- Tuples: $<8, -1>, <12, +1>, <11, -1>, <13, +1>, <14, -1>, <15, +1>$
- Sorted: $<8, -1>, <11, -1>, <12, +1>, <13, +1>, <14, -1>, <15, +1>$

**The algorithm**

- Best = 0, count = 0
- for all tuples `time[i], type[i]`:
  - `count = count + type[i]`
  - if(count > best) {
    - `best = count`
    - `beststart = time[i]`
    - `bestend = time[i+1]`
  }
- return `[beststart, bestend]`

Notes:
- `count`: numbers of “active” intervals
- `best`: best numbers of “active” intervals we have seen
- `count = count + type[i]`: if it’s a start point (type = -1), increase count; else decrease it
- `if(count > best)`: if this is the highest number of active intervals we have seen, let the best interval be `[time[i], time[i+1]]`
- If the next point is a start point, it will replace this best interval
- If the next point is an endpoint, it will end this best interval

**The algorithm at work**

Sorted: $<8, -1>, <11, -1>, <12, +1>, <13, +1>, <14, -1>, <15, +1>$

Init: `best = 0`, `count = 0`
- `<8, -1>`: `count = count - (-1) = 1`
  - Is `count > best`? Yes
    - best = 1, beststart = 8, bestend = 11
- `<11, -1>`: `count = count - (-1) = 2`
  - Is `count > best`? Yes
    - best = 2, beststart = 11, bestend = 12
- `<12, +1>`: `count = count + (+1) = 1`
  - Is `count > best`? No
- `<13, +1>`: `count = count + (+1) = 0`
  - Is `count > best`? No
- `<14, -1>`: `count = count - (-1) = 1`
  - Is `count > best`? No
- `<15, +1>`: `count = count + (+1) = 0`
  - Is `count > best`? No

return `[11, 12]`

**NTP timestamps**

How to represent time?
- “Tuesday April 19th 2011, 17:55:00”?
- “20110419175500CDT”?

NTP: 64-bit UTC timestamp
- 32 bits: offset in seconds
- 32 bits: sub-second precision
- offset = #seconds since January 1, 1900

Wraps around every $2^{32}$ seconds = 136 years
First wrap-around: 2036
Solution: 128-bit timestamp. “Enough to provide unambiguous time representation until the universe goes dim”