Leader Election in a Synchronous Ring

Paulo Sérgio Almeida

Distributed Systems Group
Departamento de Informática
Universidade do Minho
Motivation: token ring networks

- In a local area ring network a token circulates around;
- Sometimes the token gets lost;
- A procedure is needed to regenerate the token;
- This amounts to electing a leader;
Leader election in a synchronous ring

The Problem

The problem

Network graph:
- $n$ nodes, 1 to $n$ clockwise;
- symmetry and local knowledge:
  - nodes do not know their or neighbor numbers;
  - distinguish clockwise and anti-clockwise neighbors.
- notation: operations mod $n$ to facilitate;

Requirement:
- eventually, exactly one process outputs the decision leader;
Versions of the problem

- The other non-leader processes must also output *non-leader*;
- The ring can be:
  - unidirectional;
  - bidirectional;
- Number of processes $n$ can be:
  - known;
  - unknown;
- Processes can be:
  - identical;
  - have totally ordered *unique identifiers* (UID);
Let $A$ be a system of $n > 1$ processes in a bidirectional ring. If all $n$ processes are identical, then $A$ does not solve the leader-election.

Proof.

Assume WLOG that we have one starting state. (A solution admitting several starting states would have to work for any of those). We have, therefore, a unique execution. By a trivial induction on $r$, the rounds executed, we can see that all processes have identical state after any number of rounds. Therefore, if any process outputs \textit{leader}, so must the others, contradicting the uniqueness requirement.

- If all processes are identical, the problem cannot be solved!
- Intuition: by symmetry, what one does, so do the others;
Breaking symmetry

- Impossibility follows from symmetry;
- Must break symmetry; e.g. with unique UIDs;
- Symmetry breaking is an important part of many problems in distributed systems;
A basic algorithm – LCR

- LCR algorithm (Le Lann, Chang, Roberts);
- Uses comparisons on UIDs;
- Assumes only unidirectional ring;
- Does not rely on knowing the size of the ring;
- Only the leader performs output;
LCR informally

- Each process sends its UID to next;
- If a received UID is greater than self UID, it is relayed on;
- If it is smaller, it is discarded;
- If it is equal, the process outputs leader;
LCR formally

- Algorithm parameterized on process index \((i)\) and UID \((u)\);
- Message alphabet \(M = \mathbb{U}\), the set of UIDs;
- Process state, \(\text{state}_i\):
  - \(\text{send} \in M \cup \text{null}\), initially \(u\);
  - \(\text{status} \in \{\text{unknown, leader}\}\), output variable, initially \text{unknown};
- Message-generating function:
  \[
  \text{msg}_{i,u}((\text{send}, \text{status}), i + 1) = \text{send};
  \]
- State-transition function:
  \[
  \text{trans}_{i,u}((\text{send}, \text{status}), \text{msg}) = \begin{cases} 
    (\text{null}, \text{status}) & \text{if } \text{msg} = \text{null} \\
    (\text{null}, \text{status}) & \text{if } \text{msg} < u \\
    (\text{msg}, \text{status}) & \text{if } \text{msg} > u \\
    (\text{null, leader}) & \text{if } \text{msg} = u
  \end{cases}
  \]
Proof of correctness

Let $m$ be the index of process with maximum UID $u_m$;
Show two lemmas.

**Lemma**

Process $m$ outputs leader in round $n$.

**Lemma**

Processes $i \neq m$ never output leader.

**Theorem**

$LCR$ solves leader election.
Proof of correctness - first lemma

Lemma

Process \( m \) outputs leader in round \( n \).

Proof.

- For \( i \neq m \), if after round \( r \), \( send_{i-1} = u_m \), then in round \( r + 1 \), \( send_i = u_m \);
- For \( 0 \leq r \leq n - 1 \), after \( r \) rounds, \( send_{m+r} = u_m \);
- Node before \( m \) in ring is \( m + n - 1 \);
- After round \( n - 1 \), \( send_{m+n-1} = u_m \);
- In round \( n \), \( m \) receives \( u_m \) and outputs \textit{leader}.
Lemma

Processes \( i \neq m \) never output leader.

Proof.

- A process \( i \) can only output leader if it receives \( msg = u_i \);
- A non-null message can only be some \( u_j \), from process \( j \);
- As UIDs are unique, \( msg \) would have to originate in \( i \) and travel around the ring, including \( m \);
- But as \( u_i < u_m \), \( m \) does not relay \( msg \), sending null instead;
- Therefore, \( msg \) cannot arrive at \( i \), and \( i \) cannot output leader;
Halting and non-leader outputs

- LCR as presented does not halt;
- Processes other than leader stay in *unknown* status;
- Can be modified to halt and make others output *other*;
- When leader outputs, sends *halt* message and halts;
- When a process receives *halt*, passes it on and then halts;
- Processes that receive *halt* can output *other*;
- This transformation to halting and output in all processes is quite general, and can be applied in many scenarios;
Halting and non-leader outputs; an improvement

- other processes can output \textit{other} as soon as they receive a UID greater than own;
- but they cannot halt immediately; they must keep on relaying;

Arriving at output can be sometimes much sooner than halting;
- but they are independent things;
- sometimes a premature halt, forgetting to keep on reacting, can deadlock the rest of the system;
Leader election in a synchronous ring

A basic algorithm

Halting and non-leader outputs formally

- Message alphabet: as before or \{halt\};
- Process states: as before or halted;
- Halting states: halted;
- status ∈ \{unknown, leader, other\};
- Message-generating function as before;
- State-transition function:

\[
trans_{i,u}((send, status), msg) = \begin{cases} 
  halted & \text{if } send = \text{halt} \\
  (halt, status) & \text{if } msg = \text{halt} \\
  (null, status) & \text{if } msg = \text{null} \\
  (null, status) & \text{if } msg < u \\
  (msg, other) & \text{if } msg > u \\
  (halt, leader) & \text{if } msg = u 
\end{cases}
\]
Complexity

- **Time complexity:**
  - $n$ rounds until leader elected;
  - $2n$ rounds until last process halts;
  - And if processes know the size of the ring?

- **Communication complexity:**
  - $O(n^2)$ messages in the worst case for both versions;
  - $O(n \log n)$ messages in average;
  - Which configuration results in less messages? How many?
  - Which configuration results in more messages? How many?
HS – an algorithm with $O(n \log n)$ communication complexity

- HS algorithm (Hirshberg, Sinclair);
- Uses comparisons on UIDs;
- Assumes bidirectional ring;
- Does not rely on knowing the size of the ring;
- Only the leader performs output (can be overcome with transformation);
Leader election in a synchronous ring

An algorithm with $O(n \log n)$ communication complexity

HS informally

- Processes operate in phases $l = 0, 1, 2, \ldots$;
- In each phase, processes send token with UID in both directions;
- Tokens in phase $l$ intend to travel $2^l$ and turn back to sender;
- If a received UID is greater than self UID, it is relayed on;
- If it is smaller, it is discarded;
- If it is equal, the process outputs *leader*;
Leader election in a synchronous ring

An algorithm with $O(n \log n)$ communication complexity

HS formally

- **Message alphabet:** $M = \{\text{out}\} \times \mathbb{U} \times \mathbb{N} \cup \{\text{in}\} \times \mathbb{U}$;

- **Process state, $\text{state}_i$:**
  - $s_- \in M \cup \text{null}$, initially $(\text{out}, u, 1)$;
  - $s_+ \in M \cup \text{null}$, initially $(\text{out}, u, 1)$;
  - $o \in \{\text{unknown, leader}\}$, output variable, initially unknown;
  - $l$: phase, initially 0;

- **Message-generating function:**

$$msg_{i,u}((s-, s+, o, l), j) = \begin{cases} 
    s_- & \text{if } j = i - 1 \\
    s+ & \text{if } j = i + 1 
\end{cases}$$

©2007–2010 Paulo Sérgio Almeida
Leader election in a synchronous ring

An algorithm with $O(n \log n)$ communication complexity

HS – state-transition function in imperative pseudo-code

```plaintext
s+ := null
s- := null
if message from i-1 is (out, v, h):
    case
    v > u and h > 1: s+ := (out, v, h-1)
    v > u and h = 1: s- := (in, v)
    v = u: o := leader
if message from i+1 is (out, v, h):
    case
    v > u and h > 1: s- := (out, v, h-1)
    v > u and h = 1: s+ := (in, v)
    v = u: o := leader
if message from i-1 is (in, v) and v != u:
    s+ := (in, v)
if message from i+1 is (in, v) and v != u:
    s- := (in, v)
if messages from i-1 and i+1 are both (in, u):
    l := l+1
    s+ := (out, u, 2^l)
    s- := (out, u, 2^l)
```
Problems with imperative description

- Imperative style makes it unclear the functional dependence and makes it difficult to reason;
- Different places assign to the same variable;
- Are those cases mutually exclusive?
- Examples:
  - what if messages \((\text{out}, v, 3)\) and \((\text{out}, w, 1)\) arrived at a node?
  - what if messages \((\text{out}, v, 1)\) and \((\text{in}, w)\) arrived at a node?
  - in both cases, one would have to proceed, the other turn around;
  - two different specifications for same outgoing message;
- in imperative description, the last assignment wins;
- should not happen; but won’t it? should be proven;
- Algorithm depends on some combinations of incoming messages never occurring;
Alternative: functional description

- As we need to describe functions . . .
  . . . why not adopt a functional style?
- Pseudo-code with functional flavour;
- Functions defined by cases, using pattern matching;
- Functions can be partial:
  - not all cases are covered;
  - can make functions simpler;
  - a separate proof shows those cases never happen;
  - proof would have to exist anyway, if correctness depends on it;
Leader election in a synchronous ring

An algorithm with $O(n \log n)$ communication complexity

HS formally – state-transition function

\[
\text{trans}_{i,u}((s-, s+, o, l), ((\text{out}, u, h), (\text{out}, u, h))) = \\
(\text{null}, \text{null}, \text{leader}, l)
\]

\[
\text{trans}_{i,u}((s-, s+, o, l), ((\text{in}, u), (\text{in}, u))) = \\
((\text{out}, u, 2^{l+1}), (\text{out}, u, 2^{l+1}), o, l + 1)
\]

\[
\text{trans}_{i,u}((s-, s+, o, l), (m-, m+)) \text{ when } \text{lasthop}(m-, m+) = \\
(\text{filter}_u(m-), \text{filter}_u(m+), o, l)
\]

\[
\text{trans}_{i,u}((s-, s+, o, l), (m-, m+)) = \\
(\text{filter}_u(m+), \text{filter}_u(m-), o, l)
\]

\[
\text{lasthop}((\text{out}, _, 1), _) = \text{true}
\]

\[
\text{lasthop}(_, (\text{out}, _, 1)) = \text{true}
\]

\[
\text{lasthop}(_, _) = \text{false}
\]

\[
\text{filter}_u((\text{out}, v, h)) \text{ when } v < u = \text{null}
\]

\[
\text{filter}_u((\text{out}, v, 1)) = (\text{in}, v)
\]

\[
\text{filter}_u((\text{out}, v, h)) = (\text{out}, v, h - 1)
\]

\[
\text{filter}_u(m) = m
\]
HS – correctness

- Several steps in the proof;
- Safety:
  - At most one process decides to become leader;
- Termination:
  - Some process will decide to become leader;
Lemma

A process with UID $u$ outputs leader when a message started at $u$ travels the whole ring and arrives back at $u$.

Proof.

- a process with UID $u$ only decides leader when receiving a message $m = (\text{out}, u, \_)$;
- as all UIDs are different, the message started at $u$;
- as the message is outgoing, it has not turned back and travelled always in the same direction;
- therefore, the message travelled the whole ring.
Lemma

At most one process can become leader: the one with the maximum UID.

Proof.

- from the previous lemma, for a process with UID $v$ to become leader, it must receive a message $(\text{out}, v, \_)$ that travelled the whole ring;
- such message must have been subject to the $\text{filter}_u$ function for every other process;
- the only way for the message to arrive non-null is $v$ to be greater than all other UIDs.
Lemma

Process $p$ with maximum UID $u$ decides leader in round $n + 2 \times \sum_{l=0}^{m} 2^l$, with $m$ the greatest integer such that $2^m < n$.

Proof.

- messages $(out, u, \_)$ started at $p$ are always relayed; never discarded;
- for phases $0 \leq l \leq m$, such messages are outbound $2^l$ rounds, turn around, and take another $2^l$ rounds until reaching $p$, when a new phase starts;
- in the end of round $n$ of phase $m + 1$, the outbound messages, which started with $2^{m+1} \geq n$ possible hops, reach $p$ before turning back and $p$ decides leader.
Deriving a variant of HS with smaller messages

- Can we send less information in messages?
- Algorithm operates in lockstep;
- Can we move some state that controls algorithm from messages to processes?
  - Example: number of hops in messages;
    - can we control turn around of messages with process state?
- Insight:
  - everything happens in lockstep;
  - all messages travel with the same hops left;
- Is it so? Must prove;
Deriving a variant of HS with smaller messages

Lemma

In each round, all non-null messages are either outgoing with same remaining hops left, or incoming.

Proof.

- induction on the number of rounds;
- base case: all messages \((\text{out},\_\text{,}1)\);
- inductive step: messages generated are either \text{null}, the result of \text{filter}_u(\text{)}, which preserves hypothesis, or \((\text{out},\_\text{,}2^{l+1})\);
- induction hypothesis not enough . . .
Deriving a variant of HS with smaller messages

Proof.

(continued)

need to strengthen lemma and prove also that:

Lemma

All processes that start a new phase, do it in the same round.

Proof.

• proof both lemmas together: use both lemmas in the inductive step;
• not enough: why do processes start phase in same round? …
Deriving a variant of HS with smaller messages

Proof.

(Continued)
Need to strengthen lemma and prove also that:

Lemma

*All surviving messages turn around in the same round.*

Proof.

Use the three lemmas together in the inductive step.
Deriving a variant of HS with smaller messages

- In proving insight we learned much about algorithm;
- Looks possible to control message relaying or turning back:
  - without having hops in messages;
  - without having direction in messages;
- Sketch:
  - processes count rounds in each phase;
  - half-way through a phase, invert direction of messages;
  - at end of phase check if both messages received have self UID, to decide whether sending new messages;
  - processes keep counting phases and rounds, even after stopping sending new messages;
  - improvement: non-leader output can be decided earlier;
HS variant

- Message alphabet: $M = \mathbb{U}$;
- Process state, $state_i$:
  - $s- \in M \cup \text{null}$, initially $u$;
  - $s+ \in M \cup \text{null}$, initially $u$;
  - $o \in \{\text{unknown, nonleader, leader}\}$, output variable, initially unknown;
  - $l$: phase, initially 0;
  - $r$: round in phase, initially 1;
- Message-generating function:
  \[
  msg_{i,u}((s-, s+, o, l), j) = \begin{cases} 
  s- & \text{if } j = i - 1 \\
  s+ & \text{if } j = i + 1 
  \end{cases}
  \]
HS variant – state-transition function

\[ \text{trans}_{i,u}((s-, s+, o, l, r), (m-, m+)) \text{ when } (r = 2^l) = \]
\[ (\text{filter}_u(m-), \text{filter}_u(m+), o, l, r + 1) \]
\[ \text{trans}_{i,u}((s-, s+, o, l, r), (u, u)) \text{ when } (r = 2 \times 2^l) = \]
\[ (u, u, o, l + 1, 1) \]
\[ \text{trans}_{i,u}((s-, s+, o, l, r), (m-, m+)) \text{ when } (r = 2 \times 2^l) = \]
\[ (\text{null}, \text{null}, \text{nonleader}, l + 1, 1) \]
\[ \text{trans}_{i,u}((s-, s+, o, l, r), (u, u)) = \]
\[ (\text{null}, \text{null}, \text{leader}, l, r + 1) \]
\[ \text{trans}_{i,u}((s-, s+, o, l, r), (m-, m+)) = \]
\[ (\text{filter}_u(m+), \text{filter}_u(m-), o, l, r + 1) \]

\[ \text{filter}_u(v) \text{ when } v < u = \text{null} \]
\[ \text{filter}_u(m) = m \]
• **Time complexity:**
  - leader in round $n + 2 \times \sum_{i=0}^{m} 2^i$, with $m = \lceil \log_2 n \rceil - 1$;
  - $O(n)$, at most $5n$;

• **Communication complexity:**
  - a process sends new messages in phase $l$ if receives both messages from phase $l - 1$;
  - messages must have survived $2^{l-1}$ filterings;
  - within any group of $2^{l-1} + 1$ consecutive processes, at most one sends new messages in phase $l$;
  - total number of messages during phase $l$ bounded by:

$$4 \left( 2^l \cdot \left\lfloor \frac{n}{2^{l-1} + 1} \right\rfloor \right) \leq 8n$$

• total number of messages at most $8n(1 + \lceil \log_2 n \rceil)$;
• communication complexity: $O(n \log n)$