Advanced Network Security

2. Distributed Algorithms:
   Leader Election and Mutual Exclusion

Jaap-Henk Hoepman

Digital Security (DS)
Radboud University Nijmegen, the Netherlands
http://www.cs.ru.nl/~jhh

Leader Election

Leader election: motivation

- IBM token ring (1970)
  - For local area network
  - Single token traversing ring
  - Station with token was allowed to send

- How to
  - Start the network?
    - 0 tokens
  - Recover from an error?
    - 1 token
Leader election

Given a graph $G$ of nodes, design a protocol that will elect a single node as leader.

- There is one node with $\text{leader} = \text{true}$.
- For all $x, y \in V$ we have $\text{leader}(x) = \text{false}$.

**Assumptions**
- $G$ is connected, i.e., nodes can reach each other; we assume a ring network here.
- Nodes have unique identifiers $\text{id}(x)$.
  - E.g., MAC addresses.
  - Note that nodes do NOT know $\text{id}$.

**Requirements**
- Correctness: at most one leader is elected (and once elected stays elected).
- Progress: eventually a leader is elected.
- Used to:
  - Recover from errors.
  - The leader coordinates the repair.

Some non-solutions

- Node with $\text{id}(x) = 0$ becomes leader.
  - May not exist. E.g., if identifiers are based on MAC addresses.

Consider the following protocol for node $x$:

1. Send clockwise (right) $\text{id}(x)$.
2. Receive counterclockwise (left) $\text{id}(y)$.
3. $\text{leader}(x) = (\text{id}(y) < \text{id}(x))$.

This protocol assumes that $\text{id}(y) = y$, i.e., assigned increasing along the ring. This is not necessarily the case.

LeLann’s protocol: leader election on a ring

**Assumption**
- FIFO message passing and unique identifiers.
  - Note: nodes do not know the size of the ring.

**Protocol for node $x$**

1. $L = 0$.
2. $\text{leader} = \text{false}$.
3. Send right $\text{id}(x)$.
4. While $\text{leader} = \text{false}$ do:
   - Receive left $\text{id}(y)$.
   - If $\text{id}(z) = \text{id}(y)$ and $y < x$ then:
     - $L = L + 1$.
   - $\text{leader} = (\text{id}(z) = \text{id}(x))$.

Election point.
LeLann

Why does this work?

What is the message / round complexity?

What if message passing is not FIFO?

LeLann proof of correctness (1)

Correctness: at most one leader is elected (and once elected stays elected).

- We need to prove that for all nodes i, if i reaches the election point, we have li = g. Then the result follows as nodes have unique identifiers.
- In fact we will show that we have li = \{i \in \mathbb{Z}[\phi] : \phi \notin [0, n - 1]\} nodes are numbered clockwise around the ring, and i is the number of nodes – which is unique to the number of nodes.

\[ D = \text{set of indices starting at } \phi = i \text{ and going clockwise until meeting } \phi \text{ mod } n. \]

- Define \[ \text{correctness} \]

\[ \text{Why does this work?} \]

\[ \text{What if message passing is not FIFO?} \]

\[ \text{LeLann proof of correctness (1) continued} \]

\[ \text{If } x = 0 \text{ the statement holds trivially} \]

\[ \text{For } i = 1 \text{ observe that if node } i \text{ receives } x^\prime \text{ messages, then its lefthand neighbour } j = i - 1 \text{ must have sent } x^\prime \text{ messages. It does so after receiving } 1 \text{ mod } d, \ldots, 1 \text{ mod } d \text{ in that order. Given the fact that nodes send messages in the order they receive them, prepending their own value first, node } j \text{ sends the following messages } \{1\}_{d}, \{i - 1 \text{ mod } d\}_{d}, \ldots, \{1\}_{d}\text{ in that order. This is received by } i \text{ in the same order because of the FIFO property of the message channel, i.e. } i \text{ receives } \{1\}_{d}, \ldots, \{1\}_{d}\text{ in that order as required.} \]

\[ \text{If node } j \text{ reaches the election point then } \{j\}_{d} \in \phi \]

\[ \text{This happens when the } 0 \text{th message node } i \text{ receives, i.e. } \{1\}_{d} \text{ with } k = i - 1 \text{ mod } d, \text{ equals } \{1\}_{d}. \text{ This means } k = i \text{ and in } x = n \text{ and hence, } \text{ If } x = \{1\}_{d}, \text{ it } \}

\[ \text{and } n = \{1\}_{d} = 0 \text{ and } k = 0, \text{ and hence, } \text{If } x = \{1\}_{d}, \text{ it } \}

\[ \text{and } n = \{1\}_{d} = 0 \text{ and } k = 0, \text{ and hence, } \text{If } x = \{1\}_{d}, \text{ it } \}

\[ \text{and } n = \{1\}_{d} = 0 \text{ and } k = 0, \text{ and hence, } \text{If } x = \{1\}_{d}, \text{ it } \}

\[ \text{and } n = \{1\}_{d} = 0 \text{ and } k = 0, \text{ and hence, } \text{If } x = \{1\}_{d}, \text{ it } \]
LeLann proof of correctness (2)

- Progress: eventually a leader is elected.
  - Let node \( i \) have the smallest \( \#(i) \).
  - Initially node \( i \) sends \( \#(j) \) to its right-hand neighbour.
  - This means a message \( \#(j) \) is either in transit on a link (meaning the next node will eventually receive it) or received by the node (meaning it will be sent out to the right by that node).
  - Whenever this message is sent, it moves one step closer back to node \( i \).
  - Eventually node \( i \) receives \( \#(j) \) (and sends it once more to the right) and then stops.
  - It determines that \( \#(j) = \min(\#(i)) \) and hence becomes leader as required.

What if nodes do not have unique identifiers?

- Then there exists a symmetric configuration \( C \):
  - where all nodes have the same state, and all edges have the same state.
  - I.e. either all nodes are leaders, or no node is leader.
- Starting in \( C \) let all nodes take a step (the same) in turn, then:
  - all steps are local steps (changing the local state to a new state, the same for all nodes)
  - all steps are receive actions (receiving the same message), or
  - all steps are send actions (sending the same message).
- Therefore the resulting configuration \( C' \) is again symmetric.
  - We can repeat this forever, never reaching a state where there is exactly one leader.
  - This is called a symmetry argument.

Peterson’s protocol: leader election on a ring

- Bidirectional communication
  - Nodes can send messages clockwise and anticlockwise.
- Idea: algorithm proceeds in rounds
  - First round all \( n \) nodes are active and participate.
  - If a round starts with \( n \) participants, at least \( n/2 \) and at most \( n - 1 \) will be eliminated and become passive.
  - If a round start with 1 participant, it will declare itself leader at the end of the round.
Peterson's protocol: leader election on a ring

Why does it work?

What is the message/round complexity?

What if message passing is not FIFO?

There are at most \( \log_2 n \) rounds
- Node can only survive (remain active) if both its left and right active neighbour are smaller
- Therefore at most half of the nodes can survive

In every round \( 2n \) messages are sent
- An active or passive node sends exactly 2 messages in each round

So: message complexity is at most \( 2n \log_2 n \)
Mutual exclusion

Suppose nodes occasionally need access to a shared resource, but accessing it simultaneously creates problems:
- E.g. printing a document, updating a record in shared memory

We need a protocol that allows nodes to request access, and that guarantees that such nodes eventually get exclusive access:
- Nodes call `enter()` to request access
- Nodes call `exit()` to release the resource

```
while true
    enter(); /* critical section */
    exit(); /* non critical section */
```
Mutual exclusion

- A mutual exclusion protocol has the following properties:
  - Mutual exclusion: there is at most one node in the critical section.
  - Progress: if there is at least one node enters, and the critical section is empty, then one of these nodes will eventually get access to the critical section.
  - No starvation: if a node enters, and if all nodes that get access to the critical section release it, then it will eventually get access.

Assumption:
- a fair scheduler, and
- atomic shared variables

Mutual exclusion using message passing (1)

- Using logical clocks [Lamport]
  - Every message τ carries a timestamp $t_\tau$ equal to the logical clock value $L_\tau$ of the send event $\tau$; recall this induces a total order $\Rightarrow$.
  - Every node maintains a request queue of $[\tau, i]$ pairs, initially ordered $\Rightarrow$ (i.e., node 0 initially holds the resource) ordered by $\Rightarrow$ as well.

  **(1)** To request a resource, node $i$ sends a request message to all other nodes, including itself.

  **(2)** When a node $j$ receives a request message with timestamp $\tau_\rho$ from node $i$:
  - Node $j$ adds $[\tau_\rho, i]$ to its request queue.
  - Node $j$ sends a (timestamped) acknowledgement message back to $i$.

Mutual exclusion using message passing (2)

- **(3)** To release a resource, node $i$ sends a release message to all other nodes, including itself.

- **(4)** When a node $j$ receives a release message with timestamp $\tau_\rho$ from node $i$: it removes any $[\tau', i]$ messages from its request queue.

- **(5)** A node $i$ is granted access to the resource if:
  - (i) Node $i$’s request queue contains $[\tau, i]$ ordered $\Rightarrow$ before any other elements $[\tau', j]$ in the request queue.
  - (ii) Node $i$ has received a message from all other nodes with a timestamp $\tau^* > \tau$. 
Why does this work?

- Mutual exclusion
  - Rule 1 and 2 ensure that resource requests are added to all queue's
  - 5 guarantees that a node must have learned about earlier requests before honouring its own
  - Requests are only removed from the queue when the corresponding node releases it (and sends release messages to all nodes) due to rule 3 and 4

- Progress: In fact, requests are honoured in the order in which they are made
  - Follows from the fact that requests are ordered by \( n \) and served in that order because of rule 5.i and the above

- No starvation: every request is eventually honoured
  - For every request, a node receives an acknowledgement (rule 2), hence rule 5.ii is eventually satisfied
  - If every node eventually releases the resource, then rule 3 and 4 guarantee that eventually any older timestamps are removed from the request queue

---

Mutual exclusion with shared memory: the easy solution

- Use test-and-set / read-modify-write

- Cf lamport's logical clock based algorithm

---

Mutual exclusion: first try

- Two shared variables
  - flag: written by 0 and read by 1
  - flag: written by 1 and read by 0

- Protocol

---

What can go wrong?

- What if we first set the flags, before testing their value?
Mutual exclusion: Lamport’s bakery algorithm

- Each node $i$ maintains two shared variables that it writes and that all other nodes can read
  - $\text{num}(i)$, unbounded
  - $\text{choosing}(i)$, unbounded
- Idea: take numbered ticket like in the bakery
  - Except that you have to ask everyone in the shop what their number is, and take the maximum + 1
  - And you should wait for people that haven’t picked a number yet
- Lowest number is next allowed in the critical section

```
num[i] = 0
choosing[i] = false
while true
  if (choosing[i] = true)
    num[i] = 1 + max(num[j])
    choosing[i] = false
  for j = 1
    do while choosing[j] do */ wait */
      while (num[j]) >= 0 do (num[|j|] > (num[i])) do /* wait */
      /* critical section */
      num[i] = 0

```

Proof of bakery algorithm: mutual exclusion

- **Lemma 1:** If in C.S. and there is $i$ s.t. $\text{num}(i) \neq 0$ then $(\text{num}[i], i) > (\text{num}[j], j)$
  - Before entering, first waited until $\text{choosing}(i) = false$ and then waited until either $\text{num}(i) = 0$ or $(\text{num}[i], i) > (\text{num}[j], j)$
  - If $\text{num}(i) = 0$, then by assumption that no $\text{num}(i) \neq 0$, node $i$ must have changed $\text{num}(i)$ after node $i$ read it (i.e. read $\text{num}(i) \not\rightarrow \text{write-\text{num}(i)}$). Node $i$ set its current value of $\text{num}(i)$ before reading $\text{choosing}(i)$ = false and reading $\text{num}(i)$ (i.e. write-\text{num}(i) \not\rightarrow \text{read-\text{choosing}(i)} = false \not\rightarrow \text{read-\text{num}(i)}$). But node $i$ sets $\text{choosing}(i) = true$ before reading $\text{num}(i)$ and writing $\text{num}(i)$ (i.e. write-\text{choosing}(i) = true \not\rightarrow \text{write-\text{num}(i)}$).
  - This can only happen if read $\text{choosing}(i) = false \not\rightarrow \text{write-\text{choosing}(i)} = true$. Hence write-\text{num}(i) = read-\text{num}(i) no node $k$ must have seen this value for $\text{num}(i)$ when computing a ticket. By the protocol $k$ sets $\text{num}(k)$ to a larger value
  - If $(\text{num}[i], i) > (\text{num}[j], j)$ then either node $k$ did not change the value, or it entered again and by the same argument as above sets $\text{num}(k)$ to a larger value.
Proof of bakery algorithm: mutual exclusion

**Lemma 1:** If \( i \) is in C.S. and there is a \( i' \) s.t. \( \text{num}(i') \neq 0 \) then \( \text{num}(i') > \text{num}(i) \).

**Lemma 2:** For all \( i' \), \( \text{num}(i') \geq 0 \).
- Follows from the protocol.

**Lemma 3:** If \( i \) is in C.S. then \( \text{num}(i) \geq 0 \).
- Follows from Lemma 2 and the fact that \( i \) chooses \( \text{num}(i) = 1 + \max_{i'} \text{num}(i') \).

**Theorem:** The bakery protocol satisfies mutual exclusion.
- Suppose not. Then for \( i \), we have \( \text{num}(i) > 0 \) and \( \text{num}(j) > 0 \) by Lemma 3 and then both \( \text{num}(i) > \text{num}(j) \) and \( \text{num}(j) > \text{num}(i) \) by Lemma 1. A contradiction.

Proof of bakery algorithm: progress

- Suppose C.S. is empty and there are nodes entering.
- Note: the exit() protocol will eventually be completed.
- Every node that enters gets a ticket.
- Let \( i \) be the node with minimal \( (\text{num}(i), i) \).
- It will never see nodes with smaller tickets.
- It only needs to wait for nodes that choose, but these always complete after some number of steps (non-blocking operation).
- It eventually enters the C.S., so we have progress.

Proof of bakery algorithm: no starvation

- In general we have that if progress holds, and if any node \( i \) is waiting at most a finite number of other nodes can enter the C.S., then no starvation holds.
- Lemma 4: Let \( i \) have a ticket. Any node entering after that will have a bigger ticket. Hence at most \( n-i \) nodes can have a smaller ticket and beat node \( i \).
- Because progress holds and lemma 4 holds, no starvation then also holds.