Network Algorithms

More Leader Elections: Consensus with Failures!

Slides: Maurice Herlihy, Costas Busch, Roger Wattenhofer
Leader election solved!
So we can solve any network problem!? 
Game/Lecture over?
Recap (2)

- **Leader Election and Consensus**
  - When possible, when not? It depends!

- **Leader Election in Networks**
  - Not possible in ring without IDs
  - With IDs, in $O(n)$ time and $O(n \log n)$ messages, even if $n$ unknown and transmissions asynchronous
  - This is optimal: in asynchronous, need $> n \log n$ message is ring!
  - *Synchronous*: can do with $n$ messages, treat messages with time!
• Introducing failures: Byzantine Generals consensus impossible under message loss
  – Harder than NP-hard 😊

• Make it simpler: consensus in shared memory (no network!)
  – Consensus possible if processes do not die! Your algo: write value in my register, decide on minimum!
  – What about failures, does it work too?! If failures are fail-stop? If behavior is “malicious”? If behavior is Byzantine (arbitrary)?
Sequential Computation

memory

object

object

thread
Parallel Computation

memory

object

object
Often simpler than thinking about networks! “Higher-level language”, can focus on fundamental distributed system aspects!
Asynchrony

- Sudden unpredictable delays
  - Cache misses (*short*)
  - Page faults (*long*)
  - Scheduling quantum used up (*really long*)
Model

- Multiple *threads*
  - Sometimes called *processes*
- Single shared *memory*
- *Objects* live in memory
- Unpredictable asynchronous delays
Two Generals Problem

Red army wins if both sides attack together
Red armies send messengers across valley
Communications

Messengers don’t always make it
There is no non-trivial protocol that ensures the red armies attack simultaneously!
Proof Strategy

- Assume a protocol exists
- Reason about its properties
- Derive a contradiction
1. Consider the protocol that sends fewest messages
2. It still works if last message lost
3. So just don’t send it
   – Messengers’ union happy
4. But now we have a shorter protocol!
5. Contradicting #1
Consensus: Start...
... Communicate...
... Agree on Someone’s Input!
Why Consensus?

• With consensus, you can implement anything you can imagine…

• Examples: with consensus you can decide on a leader, implement mutual exclusion, or solve the two generals problem
What you will learn...

- In some models, consensus is possible
- In some other models, it is not

- Goal of this and next lecture: to learn whether for a given model consensus is possible or not … and prove it!
Problem:

- $n$ processors, with $n > 1$
- Processors can *atomically read or write* (not both) a shared memory cell
- Must decide on one of the input values

Idea:

- There is a designated memory cell $c$.
- Initially $c$ is in a special state “?”
- Processor 1 writes its value $v_1$ into $c$, then decides on $v_1$.
- A processor $j$ ($j$ not 1) reads $c$ *until* $j$ reads something else than “?”, and then decides on that.
Problems: Unexpected Delay ...
Problems: ... Heterogenous Resources ...
Problems: ... Fault-Tolerance?

Keeps lock on objects…
Problems: ... Fault-Tolerance?

E.g., your algorithm:
when to decide on minimum? And minimum of which subset of processes?
Asynchronous! Has other process died??
Need to be “wait-free”!

Keeps lock on objects…
Consensus #2: Wait-Free Shared Memory

- n processors, with n > 1
- Processors can atomically read or write (not both) a shared memory cell
- Processors might crash (halt)
- Wait-free implementation… huh?
Model

- **Wait-free** = every process (method call) completes in a finite number of its own steps
  - if scheduled sufficiently frequently, we’re fine
- **Register**: object that supports read/write
  - not punctual, takes time!
- **We assume that we have wait-free atomic register implementations**
  - that is, it seems that reads and writes to same register do not overlap, and the real-time order is respected
  - real-time: response of previous operation precedes the invocation of the next operations
  - “linearizable”
• There is a cell $c$, initially $c=\text{"}\text{?}\text{"}\text{"}$
• Every processor $i$ does the following

$$
r = \text{Read}(c);$$

$$
\text{if } (r == \text{"}\text{?}\text{"}\text{"}) \text{ then}
\begin{align*}
& \text{Write}(c, v_i); \text{ decide } v_i; \\
\end{align*}
$$

$$
\text{else}
\begin{align*}
& \text{decide } r;
\end{align*}
$$
Correct?

Stefan Schmid @ T-Labs Berlin, 2013/4
Proof Strategy:

- Make it simple
  - $n = 2$, binary input
  - one or more r/w registers
- Assume that there is a protocol (choose yours!)
- Reason about the properties of any such protocol
- Derive a contradiction: choose “bad schedule”, i.e., who is next (asynchronous), who dies, …
• Either A or B “moves” (atomic read/write!)
  – Asynchronous: “scheduler can choose!”

• Moving means
  – Register read
  – Register write
The 2-Move Tree
Decision Values
Bivalent: Both Possible

bivalent

Diagram showing a tree structure with nodes labeled 0 and 1.
Univalent: Only One Possible

Diagram showing a computational process with univalent outcomes.
Univalent: Only One Possible

univalent

0-valent

1-valent

1
0
0
1
1
0
Summary

- Wait-free computation is a tree
- Bivalent system states
  - Outcome not fixed, even *given the process inputs* (but not the execution / who dies)
- Univalent states
  - Outcome is fixed
  - Maybe not “known” yet
  - 1-Valent and 0-Valent states
Claim

Exists bivalent system state.

(The outcome is not always fixed from the start, even if process values are given.)
0-Valent Initial State

All executions lead to decision of 0
0-Valent Initial State

All executions lead to decision of 0
Solo execution by A also decides 0
1-Valent Initial State

All executions lead to decision of 1
Solo execution by B also decides 1

1-Valent Initial State
Can all executions lead to the same decision? No, must depend on execution!
Bivalent Initial State

Solo execution by A must decide 0

Solo execution by B must decide 1
Definition: Critical State

0-valent

1-valent

critical
Critical States

- Starting from a bivalent initial state
- The protocol can reach a critical state
  - Otherwise we could stay bivalent forever
  - And the protocol is not wait-free
  - Note: if only left 0-valent, right still bivalent: not critical yet, take c lower

If A goes first, protocol decides 0
If B goes first, protocol decides 1
Critical States

• Starting from a bivalent initial state
• The protocol can reach a critical state
  – Otherwise we could stay bivalent forever
  – And the protocol is not wait-free
  – Note: if only left 0-valent, right still bivalent: not critical yet, take c lower

We will show that processes cannot distinguish who goes first: Contradiction to critical state!

If A goes first, protocol decides 0
If B goes first, protocol decides 1
• So far, memory-independent!
• True for
  – Registers
  – Message-passing
  – Carrier pigeons
  – Any kind of asynchronous computation
What Are the Threads Doing?

- Reads and/or writes
- To same/different registers (one or more registers!)

Possible Interactions after critical state:

<table>
<thead>
<tr>
<th></th>
<th>x.read()</th>
<th>y.read()</th>
<th>x.write()</th>
<th>y.write()</th>
</tr>
</thead>
<tbody>
<tr>
<td>x.read()</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>y.read()</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>x.write()</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>y.write()</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
Reading Registers: Interaction of «Read»??

A runs solo, decides 0

B reads x

C

A runs solo, decides 1

States look the same to A

A cannot learn that B read x…
### Possible Interactions

<table>
<thead>
<tr>
<th></th>
<th>x.read()</th>
<th>y.read()</th>
<th>x.write()</th>
<th>y.write()</th>
</tr>
</thead>
<tbody>
<tr>
<td>x.read()</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>y.read()</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>x.write()</td>
<td>no</td>
<td>no</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>y.write()</td>
<td>no</td>
<td>no</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
A writes $y$

B writes $x$

A and B cannot distinguish: register $x=0$, register $y=1$ anyway!

Contradiction to critical state, remains bivalent
### Possible Interactions

<table>
<thead>
<tr>
<th></th>
<th>\texttt{x.read()}</th>
<th>\texttt{y.read()}</th>
<th>\texttt{x.write()}</th>
<th>\texttt{y.write()}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{x.read()}</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>\texttt{y.read()}</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>\texttt{x.write()}</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>\texttt{y.write()}</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>
States look the same to A

A runs solo, decides 0

A writes x

B writes x

A runs solo, decides 1

A writes x
That’s all, folks!

<table>
<thead>
<tr>
<th></th>
<th><code>x.read()</code></th>
<th><code>y.read()</code></th>
<th><code>x.write()</code></th>
<th><code>y.write()</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x.read()</code></td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td><code>y.read()</code></td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td><code>x.write()</code></td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td><code>y.write()</code></td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>
Theorem

- It is impossible to solve consensus using read/write atomic registers
  - Assume protocol exists
  - It has a bivalent initial state
  - Must be able to reach a critical state
  - Case analysis of interactions
    - Reads vs others
    - Writes vs writes
What does Consensus have to do with Distributed Systems?
We want to build a concurrent FIFO queue...
... with multiple dequeuers.
A Consensus Protocol based on 2 FIFO Queues & Atomic Registers

2-element array

FIFO Queue with red and black balls

Coveted red ball (winner)

Dreaded black ball (looser)
Rough Protocol Idea

1. Put my value to “my” register
2. Dequeue initialized queue: am I winner or loser?
Write Value to Array
Protocol: Take Next Item from the Queue
The Protocol

I got the coveted red ball, so I will decide my value

I got the dreaded black ball, so I will decide the other’s value from the array
Why does it work?

• If one thread gets the red ball
• Then the other gets the black ball
• Winner can take her own value
• Loser can find winner’s value in array
  – Because threads write array before dequeuing from queue