Distributed Algorithms

Johannes Åman Pohjola
UNSW
Term 2 2021
Where we’re at

We’ve concluded our coverage of proof methods, and dipped our toes into process algebra.

This week, we’ll discuss some classic distributed algorithms.

First up though...
The final exam will start on August 24 8AM–August 25 8AM.

It’s a 3–4h exam with a 24h timing window. This means you control your own scheduling: break for lunch, go to the beach, sleep on it and try again in the morning...

I’ll email you the exam papers when the exam starts. Submission is via give, same as homework and assignments.

I’ll talk about the content of the exam in Week 10.
Parallel Distributed Execution

P

- CPU
  - L1 Cache
  - L2 Cache
  - Main Memory

P'

- CPU
  - L1 Cache
  - L2 Cache
  - Main Memory

Q

- CPU
  - L1 Cache
  - L2 Cache
  - Main Memory

R

- CPU
  - L1 Cache
  - L2 Cache
  - Main Memory

network
Parallel Distributed Execution

Computation can be distributed over several *nodes* (or *locations*). Communication between nodes uses message passing. Ben-Ari’s basic model is: reliable asynchronous message passing with possible reordering of messages.

Locally, each node may run several *processes*. Processes on the same node communicate via shared memory.

**NB**

For convenience, we will generally assume that all local computation at a node is executed atomically. (We know how to do that already.)

“In particular, when a message is received the handling of the message is considered part of the same atomic statement.” - Ben-Ari
Sending and Receiving Messages

send(tag, destination, [parameters])
receive(tag, [parameters])

node 5
integer k ← 20
send(request, 3, k, 30)

node 3
integer m, n
receive(request, m, n)

Senders are anonymous by default. Messages can be chosen based on pattern matching on the tag.
Sending a Message and Expecting a Reply

Senders can reveal their identity by passing their ID as a parameter. (But how do we know they’re not lying?)
Imagine a dumb peripheral such as an old printer on a network. The other nodes need to sort out mutually exclusive access, to avoid printing interleaved text.

This is easy if we nominate one central node as sole arbiter of who gets access. But in distributed systems, *symmetric* solutions, where no one node is indispensable, are preferred.
### Algorithm 2.1: Ricart-Agrawala algorithm (outline)

<table>
<thead>
<tr>
<th>integer myNum ← 0, set of node IDs deferred ← ∅</th>
</tr>
</thead>
</table>

#### main

**p1:** non-critical section

**p2:** myNum ← chooseNumber

**p3:** for all *other* nodes N

**p4:** send(request, N, myID, myNum)

**p5:** await replies from all *other* nodes

**p6:** critical section

**p7:** for all nodes N in deferred

**p8:** remove N from deferred

**p9:** send(reply, N, myID)

#### receive

<table>
<thead>
<tr>
<th>integer source, reqNum</th>
</tr>
</thead>
</table>

**p10:** receive(request, source, reqNum)

**p11:** if reqNum < myNum

**p12:** send(reply, source, myID)

**p13:** else add source to deferred
RA Algorithm (1)
RA Algorithm (2)

Distributed Programs
Distributed CSs
Distributed CSs #2

RA Algorithm (2)

![Diagram of RA Algorithm (2)](image)

1. Aaron
2. Chloe
3. Becky
4. Aaron, Chloe
5. Chloe

 reply

 reply

 reply

 reply
Virtual Queue in the RA Algorithm

Becky ↔ Aaron ↔ Chloe
RA Algorithm (3)
RA Algorithm (4)
There are three distinct problems with the RA algorithm sketch:

- **deadlock** when equal ticket numbers are chosen
- ¬**mutex** when low numbers are chosen later
- **deadlock** when nodes retire
Equal Ticket Numbers

Standard fix: (ab)use process IDs to break ties eg by using $\langle_{\text{lex}}$ on number/process ID pairs rather than $<$ in line p11.
Choosing Ticket Numbers

Standard fix: keep track of highest seen ticket number; choose higher than that in line p2.
Quiescent Nodes

**Standard fix:** have an *intent* flag; ignore ticket number in the absence of intent (line p11).
**Algorithm 2.2: Ricart-Agrawala algorithm**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>integer myNum ← 0</code></td>
<td>Initial value of myNum</td>
</tr>
<tr>
<td><code>set of node IDs deferred ← ∅</code></td>
<td>Initial set of deferred IDs</td>
</tr>
<tr>
<td><code>integer highestNum ← 0</code></td>
<td>Initial value of highestNum</td>
</tr>
<tr>
<td><code>boolean requestCS ← false</code></td>
<td>Initial state of requestCS</td>
</tr>
</tbody>
</table>

**Main**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop forever</td>
<td>Loop forever</td>
</tr>
<tr>
<td>p1:</td>
<td>non-critical section</td>
</tr>
<tr>
<td>p2:</td>
<td><code>requestCS ← true</code></td>
</tr>
<tr>
<td>p3:</td>
<td><code>myNum ← highestNum + 1</code></td>
</tr>
<tr>
<td>p4:</td>
<td>for all other nodes N</td>
</tr>
<tr>
<td>p5:</td>
<td>send(request, N, myID, myNum)</td>
</tr>
<tr>
<td>p6:</td>
<td>await replies from all other nodes</td>
</tr>
<tr>
<td>p7:</td>
<td>critical section</td>
</tr>
<tr>
<td>p8:</td>
<td><code>requestCS ← false</code></td>
</tr>
<tr>
<td>p9:</td>
<td>for all nodes N in deferred</td>
</tr>
<tr>
<td>p10:</td>
<td>remove N from deferred</td>
</tr>
<tr>
<td>p11:</td>
<td>send(reply, N, myID)</td>
</tr>
</tbody>
</table>
Algorithm 2.2: Ricart-Agrawala algorithm (continued)

<table>
<thead>
<tr>
<th>Receive</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>integer source, requestedNum</td>
<td></td>
</tr>
<tr>
<td>loop forever</td>
<td></td>
</tr>
<tr>
<td>p1: receive(request, source, requestedNum)</td>
<td></td>
</tr>
<tr>
<td>p2: highestNum ← max(highestNum, requestedNum)</td>
<td></td>
</tr>
<tr>
<td>p3: if not requestCS or (requestedNum,source) &lt;&lt;lex (myNum,myID)</td>
<td></td>
</tr>
<tr>
<td>p4: send(reply, source, myID)</td>
<td></td>
</tr>
<tr>
<td>p5: else add source to deferred</td>
<td></td>
</tr>
</tbody>
</table>
Correctness of RA

We show mutual exclusion and eventual entry. For mutual exclusion, suppose nodes $i$ and $k$ are in the CS; we distinguish 3 cases of when their ticket numbers, $myNum_i$ and $myNum_k$ were last chosen:

- **Case 1:** node $k$ chose $myNum_k$ after replying to $i$
- **Case 2:** node $i$ chose $myNum_i$ after replying to $k$ (symmetric)
- **Case 3:** nodes $i$ and $k$ chose $myNum_i$ and $myNum_k$ before replying
Mutual Exclusion, Case 1

*Happens-before* diagram based on local order and receive-after-send causality in this case:

\[ \begin{align*}
    i & \quad \text{choose} \quad \text{send request} \\
    k & \quad \text{receive request} \quad \text{reply} \quad \text{choose}
\end{align*} \]

\( myNum_k \) must be greater than \( myNum_i \); hence \( i \) won’t reply before leaving the CS.
Mutual Exclusion, Case 3

\[ i \text{ main} \rightarrow \text{choose} \rightarrow \text{send request} \]
\[ i \text{ receive} \]
\[ k \text{ main} \rightarrow \text{choose} \rightarrow \text{send request} \]
\[ k \text{ receive} \]

\[ \lessdot_{\text{lex}} \text{ is a total order and both } i \text{ and } k \text{ have requestCS} = \text{true, hence one of them must defer its reply.} \]
Alternative proof

This informal proof was based on *behavioural reasoning*: a style of argumentation that tends to go “if this happened then that must have happened”.

If you find such proofs a bit dodgy (in which case you’re in good company), there’s a proper formal invariant proof here:

RA: Eventual Entry

Suppose node $i$ wants to enter the CS. It will eventually progress until it’s stuck in p6, waiting for replies.

Its request messages will eventually arrive at all other nodes, making them aware of $myNum_i$. Thus, the others subsequently choose higher numbers.

As usual, nodes can only fall asleep in the non-CS, so all those ahead of $i$ in the virtual queue must eventually enter their CS and leave it, too.
Channels in RA (Promela)

Every node has a single channel for receiving messages; all senders share it.

RA promela code available on the course website.
Ricart-Agrawala works (mutex, dlf, starvation-freedom) but exchanges $2(n + 1)$ messages per CS access, even in the absence of contention.

**Idea:** have 1 *token* in the system; pass it around as a right to enter CS. We expect:

- **mutual exclusion:** trivial
- **absence of unnecessary delay:** trivial
- **deadlock-freedom:** maybe
- **starvation-freedom:** maybe not
### Algorithm 2.3: Ricart-Agrawala token-passing algorithm

| boolean haveToken ← true in node 0, false in others |
| integer array[NODES] requested ← [0, ..., 0] |
| integer array[NODES] granted ← [0, ..., 0] |
| integer myNum ← 0 |
| boolean inCS ← false |

#### sendToken

- if ∃ N. requested[N] > granted[N]
  - for some such N
    - send(token, N, granted)
  - haveToken ← false
### Algorithm 2.3: Ricart-Agrawala token-passing algorithm (continued)

<table>
<thead>
<tr>
<th>Main</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop forever</td>
</tr>
<tr>
<td>p1: non-critical section</td>
</tr>
<tr>
<td>p2: if not haveToken</td>
</tr>
<tr>
<td>p3: ( \text{myNum} \leftarrow \text{myNum} + 1 )</td>
</tr>
<tr>
<td>p4: for all other nodes ( N )</td>
</tr>
<tr>
<td>p5: send(request, ( N ), myID, myNum)</td>
</tr>
<tr>
<td>p6: receive(token, granted)</td>
</tr>
<tr>
<td>p7: ( \text{haveToken} \leftarrow \text{true} )</td>
</tr>
<tr>
<td>p8: ( \text{inCS} \leftarrow \text{true} )</td>
</tr>
<tr>
<td>p9: critical section</td>
</tr>
<tr>
<td>p10: ( \text{granted}[\text{myID}] \leftarrow \text{myNum} )</td>
</tr>
<tr>
<td>p11: ( \text{inCS} \leftarrow \text{false} )</td>
</tr>
<tr>
<td>p12: sendToken</td>
</tr>
</tbody>
</table>
Algorithm 2.3: Ricart-Agrawala token-passing algorithm (continued)

| Receive |
|-----------------|-----------------|
| Receive (request, source, reqNum) |
| requested[source] ← max(requested[source], reqNum) |
| if haveToken and not inCS |
| sendToken |
“granted” = last ticket numbers when entering CS (accurate at token owner)
“requested” = last known ticket numbers

<table>
<thead>
<tr>
<th></th>
<th>Aaron</th>
<th>Becky</th>
<th>Chloe</th>
<th>Danielle</th>
<th>Evan</th>
</tr>
</thead>
<tbody>
<tr>
<td>requested</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>granted</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>
RA Token-Passing Algorithm Properties

Only 1 token in the system $\implies$ mutex.
Requests being delivered eventually $\implies$ dlf.
Arbitrary choice of token recipient in `sendToken` $\implies$ potential starvation.

*Potential fix:* choose lowest “granted” value among those $i$ with $\text{granted}[i] < \text{requested}[i]$ as token recipient in `sendToken`.

*Remaining problem:* messages are big. Still inefficient for larger $N$. 

Neilsen-Mizuno Algorithm

\textit{Idea}: pass a token in a set of virtual trees; initially: root of a spanning tree of the system = token holder; requests are sent to the parent node; parenthood is surrendered (new root of a tree, but no token yet)
parents \textit{relay} requests from children; parenthood switched to the sender of the relayed message
token holder in CS \textit{defers} the first request until outside CS; parenthood switched to the first sender; later requests relayed as usual
Distributed System for Neilsen-Mizuno Algorithm
Spanning Tree in Neilsen-Mizuno Algorithm
Neilsen-Mizuno Algorithm (1)

(request, Aaron, Aaron)

Aaron ➔ Becky ➔ Chloe ➔ Danielle ➔ Evan

(request, Becky, Aaron)

Aaron ➔ Becky ➔ Chloe ➔ Danielle ➔ Evan

Becky changes parent

Aaron ➔ Becky ➔ Chloe ➔ Danielle ➔ Evan
Neilsen-Mizuno Algorithm (2)

Chloe: still in CS

Evan wants to enter the CS; request messages bubble up to Aaron
Neilsen-Mizuno Algorithm (3)
**Algorithm 2.4: Neilsen-Mizuno token-passing algorithm**

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer parent ← (initialized to form a tree)</td>
<td></td>
</tr>
<tr>
<td>integer deferred ← 0</td>
<td></td>
</tr>
<tr>
<td>boolean holding ← true in the root, false in others</td>
<td></td>
</tr>
</tbody>
</table>

**Main**

- loop forever
  - p1: non-critical section
  - p2: if not holding
  - p3: send(request, parent, myID, myID)
  - p4: parent ← 0
  - p5: receive(token)
  - p6: holding ← false
  - p7: critical section
  - p8: if deferred ≠ 0
  - p9: send(token, deferred)
  - p10: deferred ← 0
  - p11: else holding ← true
Algorithm 2.4: Neilsen-Mizuno token-passing algorithm (continued)

Receive

- integer source, originator
- loop forever
  - p12: receive(request, source, originator)
  - p13: if parent = 0
  - p14: if holding
  - p15: send(token, originator)
  - p16: holding ← false
  - p17: else deferred ← originator
  - p18: else send(request, parent, myID, originator)
  - p19: parent ← source
Neilsen-Mizuno: Correctness

Mutual exclusion is trivial: there’s only ever one token. The original paper has (informal, behavioural) proofs of deadlock and starvation freedom:

What now?

More distributed algorithms!

Also, Assignment 2 is out. Have a look as soon as possible!