Distributed Programs

Distributed CSs

Distributed CSs #2

COMP3151/9154

Foundations of Concurrency

Distributed Algorithms

Johannes Åman Pohjola

UNSW

Term 2 2022
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...
Exam info

The final exam will start on August 22 8AM–August 23 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
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.
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.
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.
Time, Clocks and the Ordering of Events

A fundamental problem is to reach agreement on the order of events.

We receive two messages, from other nodes in a distributed system. Which message should we treat as more “recent”? Can we use...

- the order we received them in?
- timestamps attached to messages?
A fundamental problem is to reach agreement on the order of events.

We receive two messages, from other nodes in a distributed system. Which message should we treat as more “recent”?

Can we use...

- ...the order we received them in?
- ...timestamps attached to messages?

No. Messages may arrive out-of-order. We cannot assume that the clocks at different nodes are perfectly in synch.
Time, Clocks and the Ordering of Events

Given two events from nodes A and B, node C cannot tell which happened first.

Fortunately, we don’t need to. We just need all nodes to agree on an order that could have happened; or in other words, a causally consistent order.
Time, Clocks and the Ordering of Events

Given two events from nodes A and B, node C cannot tell which happened first.

Fortunately, we don’t need to. We just need all nodes to agree on an order that could have happened; or in other words, a causally consistent order.

Remember, events in a concurrent system are partially ordered. We write \( a \rightarrow b \) ("a must happen before b") if either:

1. \( a \) and \( b \) occur in the same process, and \( a \) happens before \( b \).
2. \( a \) is the sending of a message, and \( b \) is the receipt of the same message.
3. There exists \( c \) such that \( a \rightarrow c \) and \( c \rightarrow b \) (transitivity).
Time, Clocks and the Ordering of Events

Given two events from nodes A and B, node C cannot tell which happened first.

Remember, events in a concurrent system are partially ordered. We write \( a \rightarrow b \) (“a causally depends on b”) if either:

1. \( a \) and \( b \) occur in the same process, and \( a \) happens before \( b \).
2. \( a \) is the sending of a message, and \( b \) is the receipt of the same message.
3. There exists \( c \) such that \( a \rightarrow c \) and \( c \rightarrow b \) (transitivity).

If neither of the above, \( a \) and \( b \) are concurrent events. The events we have in mind are sends and receives; we ignore internal events.
Time, Clocks and the Ordering of Events

Can we get all nodes to agree on a \textit{total} ordering of events that is consistent with $\rightarrow$ ?
Can we get all nodes to agree on a total ordering of events that is consistent with $\rightarrow$?

Lamport’s solution with logical clocks:

1. Each process $i$ maintains a logical clock $c_i \in \mathbb{N}$.
2. Each process increments $c_i$ when it performs an event.
3. When $i$ sends a message, it attaches $c_i$ (a logical timestamp).
4. When $i$ receives a message with timestamp $c_j$, assign $c_i := \max(c_i, c_j) + 1$.

Events can now be totally ordered by their timestamps! (With PIDs as tiebreakers, as in the Bakery algorithm.)
Time, Clocks and the Ordering of Events

The ordering induced by the timestamps is now causally consistent:

**Theorem (Clock condition)**

Let $C(a)$ denote the timestamp after event $a$. We have that $a \rightarrow b$ implies $C(a) < C(b)$.

More on Lamport Clocks in this classic paper:

Distributed Mutual Exclusion

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.
<table>
<thead>
<tr>
<th>Algorithm 2.1: Ricart-Agrawala algorithm (outline)</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer myNum ← 0, set of node IDs deferred ← ∅</td>
</tr>
</tbody>
</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**

| 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)
Virtual Queue in the RA Algorithm
RA Algorithm (3)

Diagram showing a network of nodes with names and values, connected by arrows indicating communication (reply).
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: abuse process IDs to break ties eg by using <lex on number/process ID pairs rather than < in line p11.
Equal Ticket Numbers

- Becky: 5
- Aaron: 5

Standard fix:
(ab)use process IDs to break ties eg by using < on number/process ID pairs rather than < in line p11.
Equal Ticket Numbers

Becky  5
Aaron

Aaron  5
Becky

deadlock
Equal Ticket Numbers

Becky  5
Aaron

Aaron  5
Becky

deadlock

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

- Becky, ticket 5
- Aaron, ticket 10

Standard fix: keep track of highest seen ticket number; choose higher than that in line.
Choosing Ticket Numbers

Standard fix:
keep track of highest seen ticket number; choose higher than that in line p2.
Choosing Ticket Numbers

Becky • 5
Aaron

Aaron 10
Choosing Ticket Numbers

- Becky: 5
- Aaron: 10

Standard fix: keep track of highest seen ticket number; choose higher than that in line p2.
Choosing Ticket Numbers

Becky

Aaron • 10

Standard fix: keep track of highest seen ticket number; choose higher than that in line p2.
Choosing Ticket Numbers

Becky: 8
Aaron: 10

Standard fix: keep track of highest seen ticket number; choose higher than that in line p2.
Choosing Ticket Numbers

Standard fix: keep track of highest seen ticket number; choose higher than that in line p2.
Choosing Ticket Numbers

Becky 8

Aaron 10

reply
Choosing Ticket Numbers

Becky • 8

Aaron • 10
Choosing Ticket Numbers

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

Becky 5

req

Aaron (zzz) 0

Standard fix: have an intent flag; ignore ticket number in the absence of intent (line p11).
Quiescent Nodes

Standard fix:

have an intent flag; ignore ticket number in the absence of intent (line p11).
Quiescent Nodes

Standard fix: have an *intent* flag; ignore ticket number in the absence of intent (line p11).
<table>
<thead>
<tr>
<th>Algorithm 2.2: Ricart-Agrawala algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer myNum ← 0</td>
</tr>
<tr>
<td>set of node IDs deferred ← ∅</td>
</tr>
<tr>
<td>integer highestNum ← 0</td>
</tr>
<tr>
<td>boolean requestCS ← false</td>
</tr>
</tbody>
</table>

**Main**

loop forever
p1:   non-critical section
p2:   requestCS ← true
p3:   myNum ← highestNum + 1
p4:   for all *other* nodes N
p5:    send(request, N, myID, myNum)
p6:    await replies from all *other* nodes
p7:    critical section
p8:    requestCS ← false
p9:    for all nodes N in deferred
p10:   remove N from deferred
p11:   send(reply, N, myID)
## Algorithm 2.2: Ricart-Agrawala algorithm (continued)

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

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

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

\(<_\text{lex} >\) is a total order and both \(i\) and \(k\) have request\(\text{CS} = \text{true}\), hence one of them must defer its reply.
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:

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.
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: myNum ← 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: haveToken ← true</td>
</tr>
<tr>
<td>p8: inCS ← true</td>
</tr>
<tr>
<td>p9: critical section</td>
</tr>
<tr>
<td>p10: granted[myID] ← myNum</td>
</tr>
<tr>
<td>p11: inCS ← false</td>
</tr>
<tr>
<td>p12: sendToken</td>
</tr>
</tbody>
</table>
### Algorithm 2.3: Ricart-Agrawala token-passing algorithm (continued)

<table>
<thead>
<tr>
<th>Receive</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer source, reqNum</td>
</tr>
<tr>
<td>loop forever</td>
</tr>
<tr>
<td>p13: receive(request, source, reqNum)</td>
</tr>
<tr>
<td>p14: requested[source] ← max(requested[source], reqNum)</td>
</tr>
<tr>
<td>p15: if haveToken and not inCS</td>
</tr>
<tr>
<td>p16: sendToken</td>
</tr>
</tbody>
</table>
Data Structures for RA Token-Passing Algorithm

“granted” = last ticket numbers when entering CS (accurate at token owner)
“requested” = last known ticket numbers

Example (Chloe’s view)

<table>
<thead>
<tr>
<th>Requested</th>
<th>4</th>
<th>3</th>
<th>0</th>
<th>5</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Granted</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

Aaron Becky Chloe Danielle Evan
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 $\text{sendToken}$ $\implies$ potential starvation.
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`.
RA Token-Passing Algorithm Properties

Only 1 token in the system \(\Rightarrow\) mutex.
Requests being delivered eventually \(\Rightarrow\) dlf.
Arbitrary choice of token recipient in \texttt{sendToken} \(\Rightarrow\) potential starvation.

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

Remaining problem: messages are big. Still inefficient for larger \(N\).
Neilsen-Mizuno Algorithm

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 relay requests from children; parenthood switched to the sender of the relayed message
token holder in CS 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

- Danielle
- Evan
- Chloe
- Aaron
- Becky
Neilsen-Mizuno Algorithm (1)

Aaron → Becky → Chloe → Danielle → Evan

Becky changes parent

Aaron → Becky

Becky → Danielle

Becky → Evan

Becky → Danielle

Becky → Evan
Neilsen-Mizuno Algorithm (1)

(request, Aaron, Aaron)

Aaron → Becky → Chloe → Danielle → Evan

Becky changes parent
Neilsen-Mizuno Algorithm (1)

(request, Aaron, Aaron)

Aaron → Becky → Chloe → Danielle → Evan

Becky changes parent

Aaron → Becky → Chloe → Danielle → Evan
Neilsen-Mizuno Algorithm (1)

(request, Aaron, Aaron)

(request, Becky, Aaron)
Neilsen-Mizuno Algorithm (1)

(request, Aaron, Aaron)

(request, Becky, Aaron)

Becky changes parent
Neilsen-Mizuno Algorithm (2)
Neilsen-Mizuno Algorithm (2)

Chloe: still in CS

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

Chloe: still in CS

- Aaron
- Becky
- Chloe
- Danielle
- Evan

Evan wants to enter the CS; request messages bubble up to Aaron
Neilsen-Mizuno Algorithm (2)

Chloe: still in CS

Evan wants to enter the CS; request messages bubble up to Aaron
Neilsen-Mizuno Algorithm (2)

Chloe: still in CS

Evan wants to enter the CS; request messages bubble up to Aaron
Neilsen-Mizuno Algorithm (3)
Neilsen-Mizuno Algorithm (3)
Neilsen-Mizuno Algorithm (3)
Neilsen-Mizuno Algorithm (3)

Distributed Programs

Distributed CSs

Distributed CSs #2

Aaron → Becky → Chloe → Danielle → Evan

(token)

Aaron → Becky → Chloe → Danielle → Evan

(token)
Neilsen-Mizuno Algorithm (3)
Algorithm 2.4: Neilsen-Mizuno token-passing algorithm

| integer parent ← (initialized to form a tree) |
| integer deferred ← 0 |
| boolean holding ← true in the root, false in others |

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!