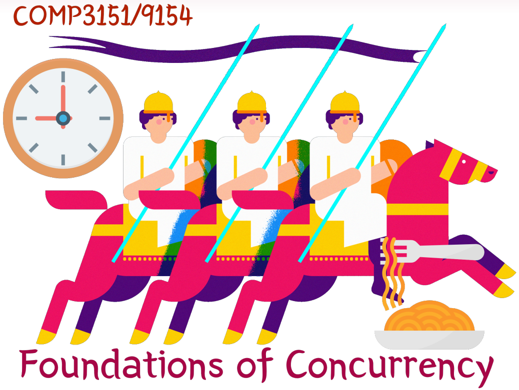


Distributed Algorithms

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

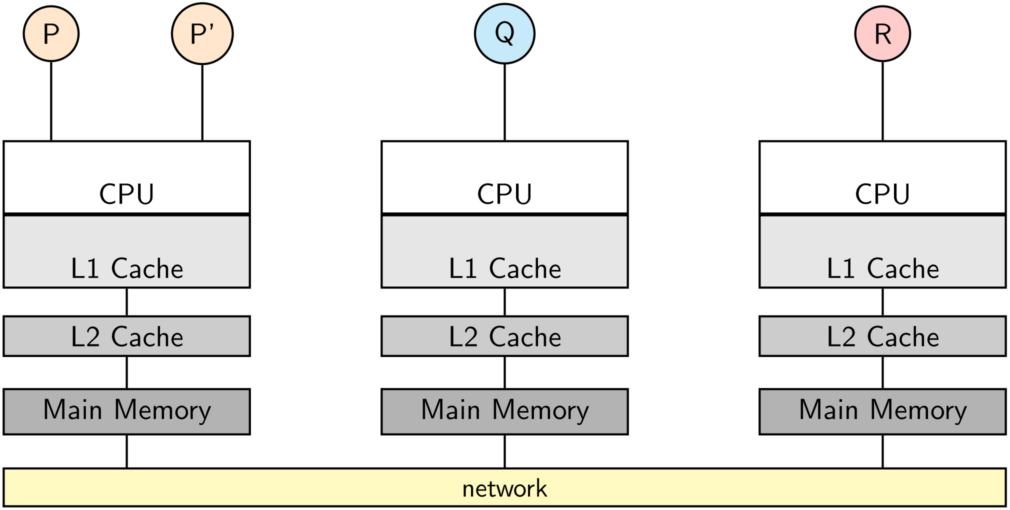
The final exam will start on August 22 8AM–August 23 8AM.

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.

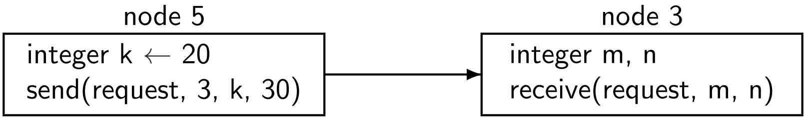
Locally, each node may run several *processes*. Processes on the same node communicate via shared memory.

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])



Senders are anonymous be 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?

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?

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 *total* ordering of events that is consistent with $\rightarrow$?

Time, Clocks and the Ordering of Events

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:

Leslie Lamport. *Time, Clocks and the Ordering of Events in a Distributed System*. CACM 1978. https://lamport.azurewebsites.net/pubs/time-clocks.pdf

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.

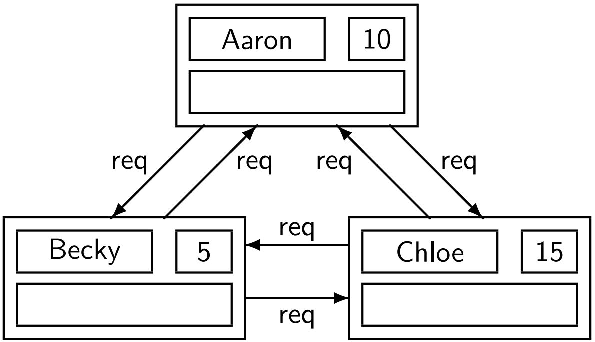
This is easy if we nominate one central node as sole arbiter of who gets access. But in distributed systems, *symmetric* solutions, where no one node is indispensable, are preferred.

Algorithm 2.1: Ricart-Agrawala algorithm (outline)

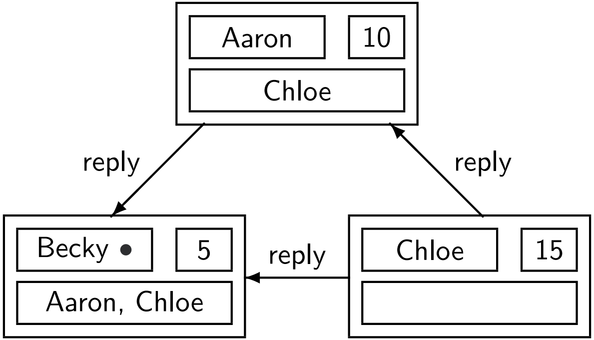
integer myNum ← 0, set of node IDs deferred ← ∅

```
     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
     integer source, reqNum
p10:    receive(request, source, reqNum)
p11:    if reqNum < myNum
p12:       send(reply,source,myID)
p13:    else add source to deferred
```

RA Algorithm (1)

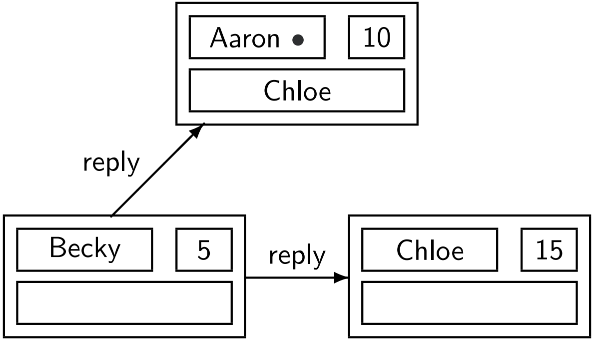


RA Algorithm (2)

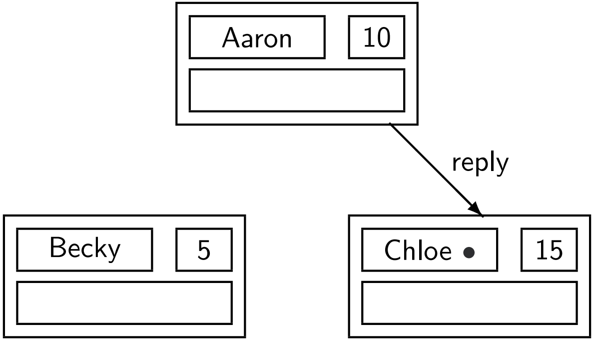


Virtual Queue in the RA Algorithm

RA Algorithm (3)



RA Algorithm (4)



Problems

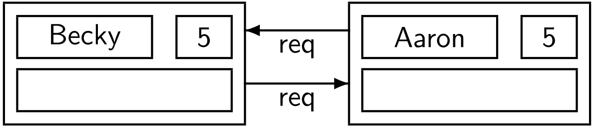
There are three distinct problems with the RA algorithm sketch:

deadlock when equal ticket numbers are chosen

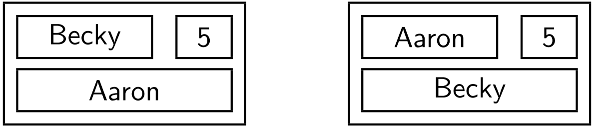
¬**mutex** when low numbers are chosen later

deadlock when nodes retire

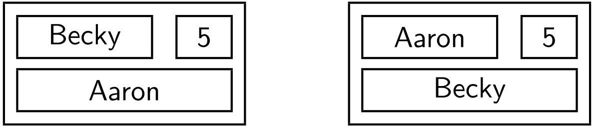
Equal Ticket Numbers



Equal Ticket Numbers

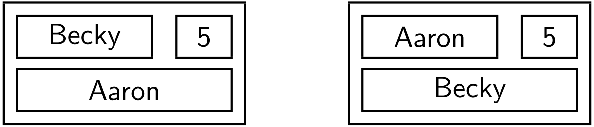


Equal Ticket Numbers



deadlock

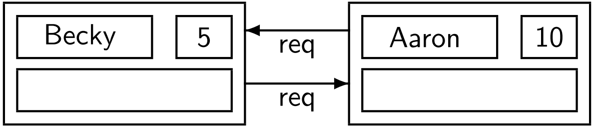
Equal Ticket Numbers



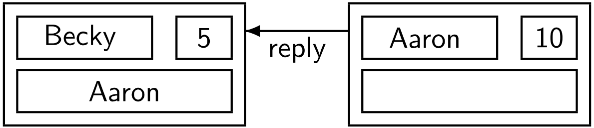
deadlock

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

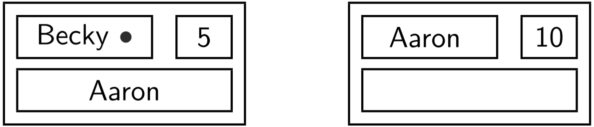
Choosing Ticket Numbers



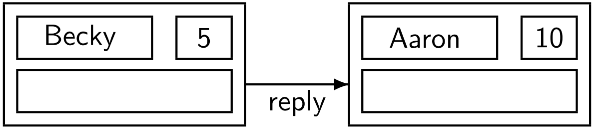
Choosing Ticket Numbers



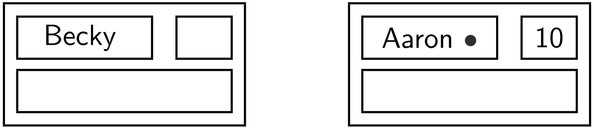
Choosing Ticket Numbers



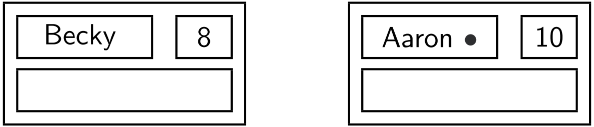
Choosing Ticket Numbers



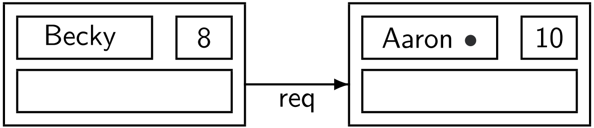
Choosing Ticket Numbers



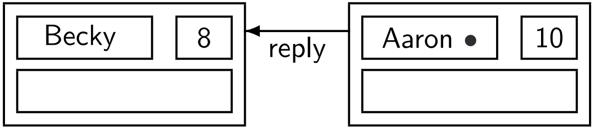
Choosing Ticket Numbers



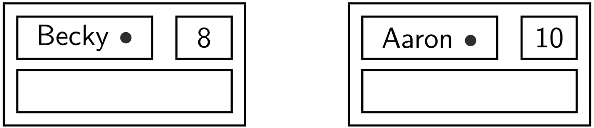
Choosing Ticket Numbers



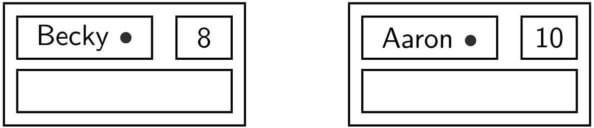
Choosing Ticket Numbers



Choosing Ticket Numbers

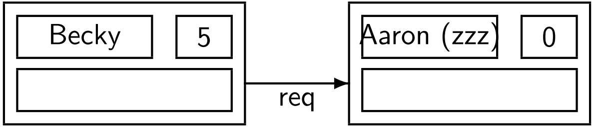


Choosing Ticket Numbers

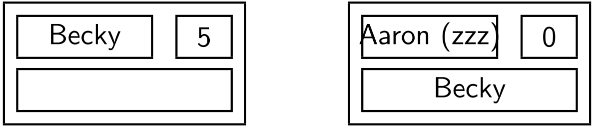


Standard fix: keep track of highest seen ticket number; choose higher than that in line p2.

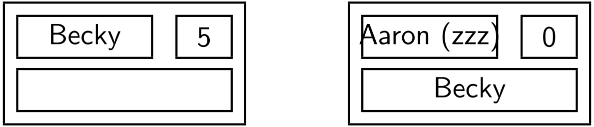
Quiescent Nodes



Quiescent Nodes



Quiescent Nodes



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

Algorithm 2.2: Ricart-Agrawala algorithm

```
                    integer myNum ← 0
                    set of node IDs deferred ← ∅
                    integer highestNum ← 0
                    boolean requestCS ← false

     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)

Receive

```
    integer source, requestedNum
    loop forever
p1:   receive(request, source, requestedNum)
p2:   highestNum ← max(highestNum, requestedNum)
p3:   if not requestCS or (requestedNum,source) <lex (myNum,myID)
p4:     send(reply, source, myID)
p5:   else add source to deferred
```

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:

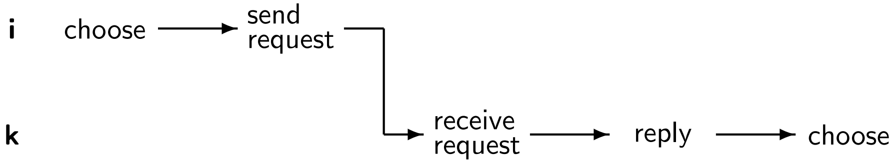
Case 1: node k chose $myNum_k$ after replying to i

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

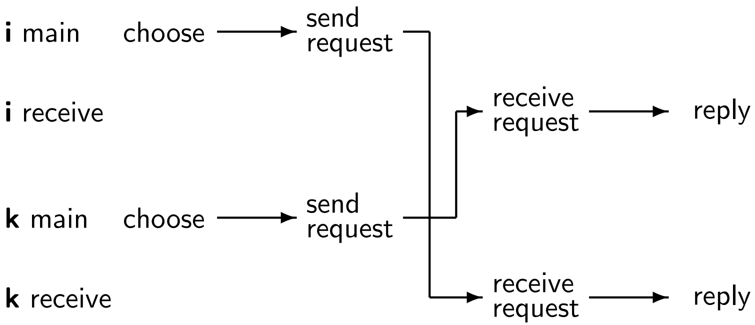
Mutual Exclusion, Case 1

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



$myNum_k$ 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 requestCS $=$ 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:

Ekaterina Sedletsky, Amir Pnueli and Mordechai Ben-Ari. *Formal Verification of the Ricart-Agrawala Algorithm*. FSTTCS 2000. https://doi.org/10.1007/3-540-44450-5_26

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.

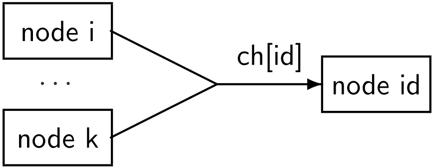
RA: Eventual Entry

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.

Back to Distributed CSs

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)

Main

```
     loop forever
p1:     non-critical section
p2:     if not haveToken
p3:        myNum ← myNum + 1
p4:        for all other nodes N
p5:           send(request, N, myID, myNum)
p6:        receive(token, granted)
p7:        haveToken ← true
p8:     inCS ← true
p9:     critical section
p10:    granted[myID] ← myNum
p11:    inCS ← false
p12:    sendToken
```

Algorithm 2.3: Ricart-Agrawala token-passing algorithm (continued)

Receive

```
      integer source, reqNum
      loop forever
p13:     receive(request, source, reqNum)
p14:     requested[source] ← max(requested[source], reqNum)
p15:     if haveToken and not inCS
p16:        sendToken
```

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)

	Aaron	Becky	Chloe	Danielle	Evan
requested	4	3	0	5	1
granted	4	2	2	4	1

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.

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 granted$[i] <$ requested$[i]$ as token recipient in **sendToken**.

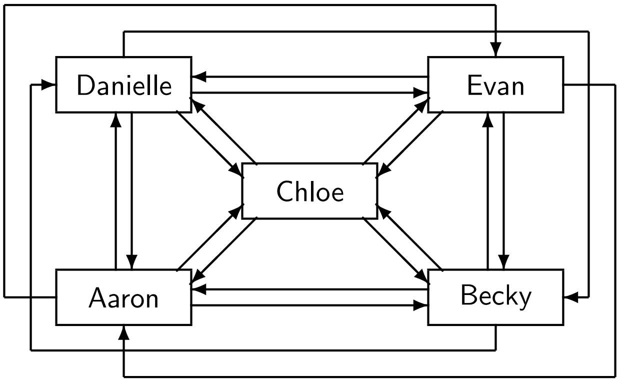
RA Token-Passing Algorithm Properties

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 granted$[i] <$ requested$[i]$ as token recipient in **sendToken**.
Remaining problem: messages are big. Still inefficient for larger N.

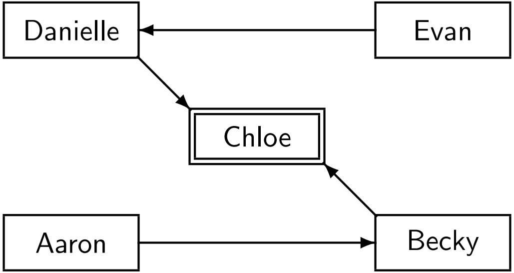
Neilsen-Mizuno Algorithm

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

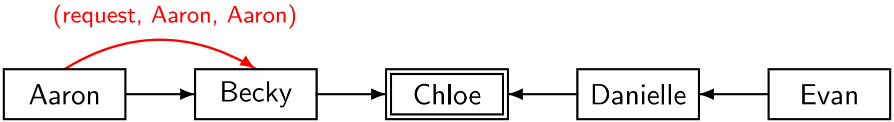


Spanning Tree in Neilsen-Mizuno Algorithm

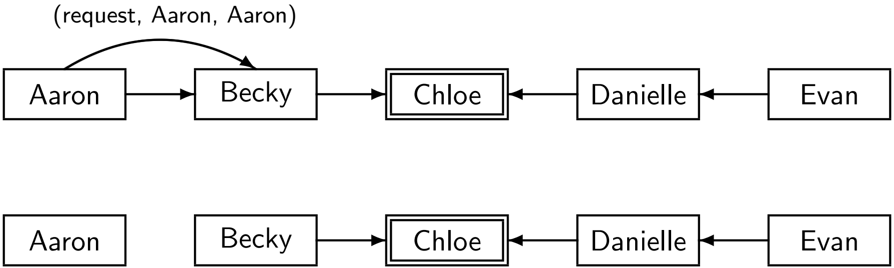


Neilsen-Mizuno Algorithm (1)

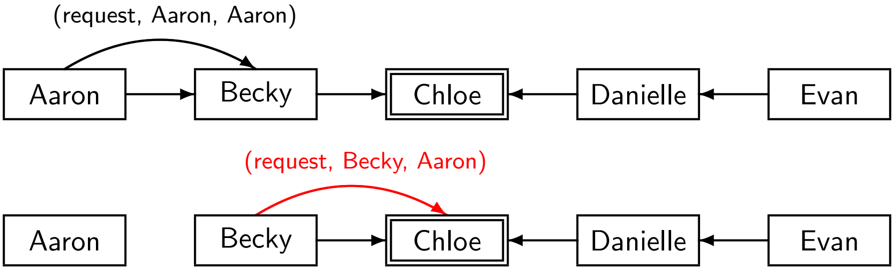
Neilsen-Mizuno Algorithm (1)



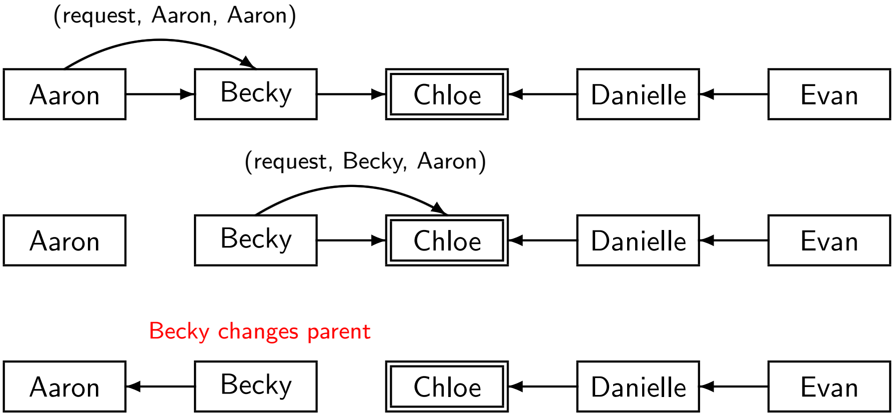
Neilsen-Mizuno Algorithm (1)



Neilsen-Mizuno Algorithm (1)



Neilsen-Mizuno Algorithm (1)



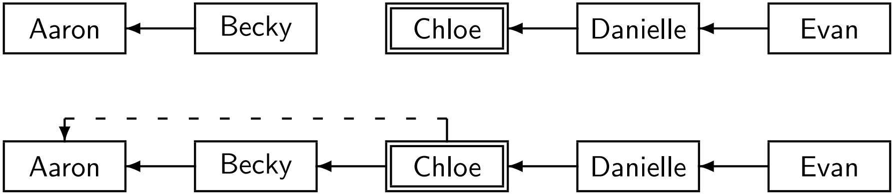
Neilsen-Mizuno Algorithm (2)

Neilsen-Mizuno Algorithm (2)

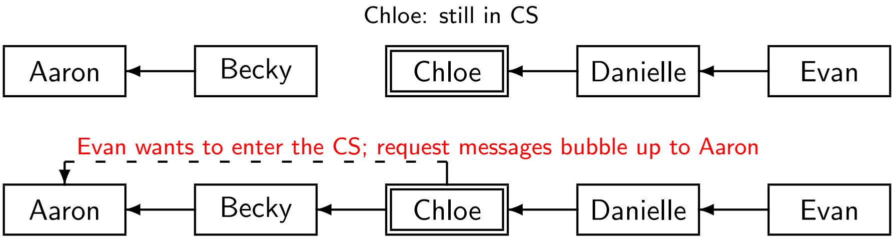
Chloe: still in CS

Neilsen-Mizuno Algorithm (2)

Chloe: still in CS



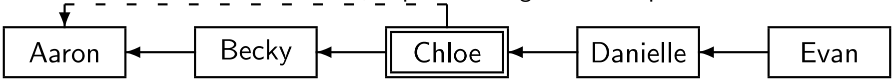
Neilsen-Mizuno Algorithm (2)

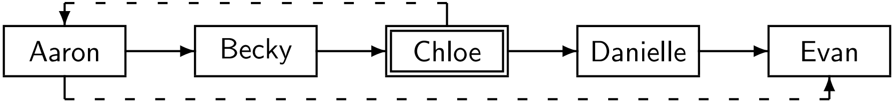


Neilsen-Mizuno Algorithm (2)

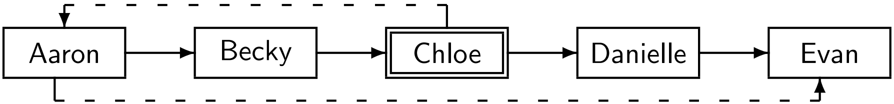
Chloe: still in CS

Evan wants to enter the CS; request messages bubble up to Aaron

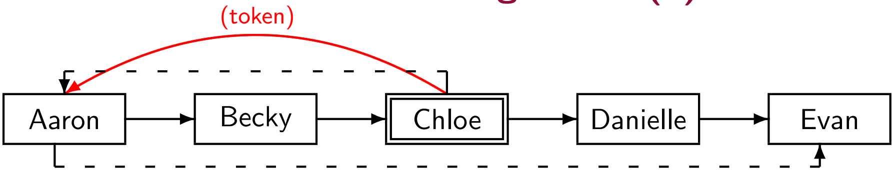




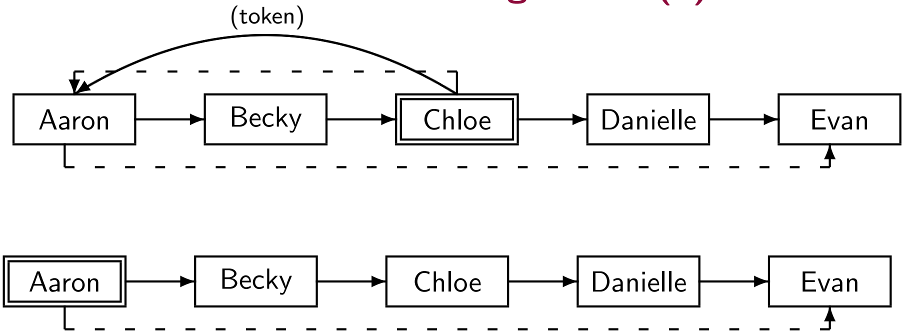
Neilsen-Mizuno Algorithm (3)



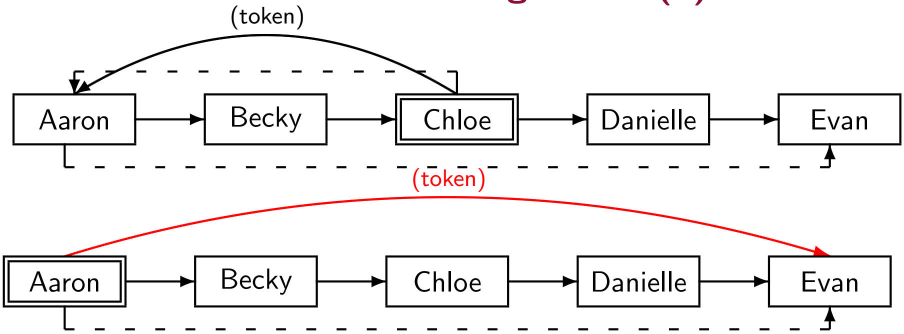
Neilsen-Mizuno Algorithm (3)



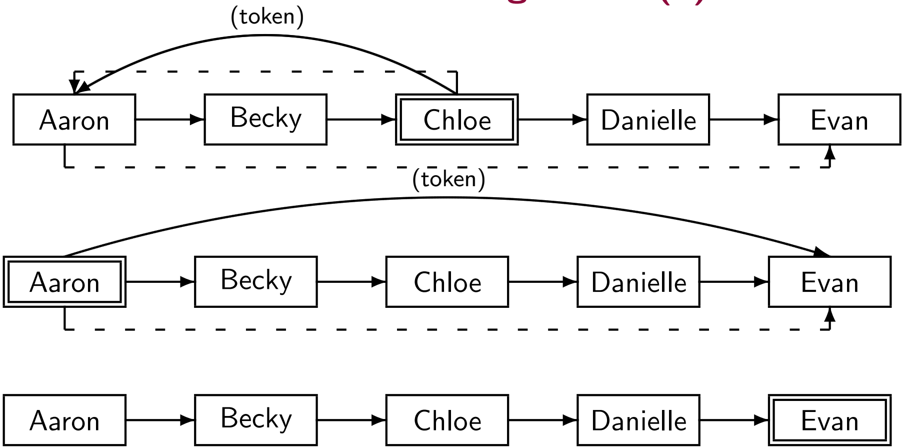
Neilsen-Mizuno Algorithm (3)



Neilsen-Mizuno Algorithm (3)



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:

Mitchell L. Neilsen and Masaaki Mizuno. *A Dag-Based Algorithm for Distributed Mutual Exclusion*. ICDCS 1991. https://doi.org/10.1109/ICDCS.1991.148689

What now?

More distributed algorithms!

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