159.355 Concurrent Systems

Hans W. Guesgen

Based on slides provided with:

Mordechai (Moti) Ben-Ari
Principles of Concurrent and Distributed Programming (SE)
Addison-Wesley, 2006

`http://www.weizmann.ac.il/sci-tea/benari/`

Distributed Algorithms

- The distributed systems model distinguishes between nodes and processes:
  - A node represents a physically identifiable object like a computer.
  - Individual computers may be running mulitple processes.
- There are two statements for communications:
  - **send(MessageType, Destination[, Parameters])**
  - MessageType identifies the type of message sent from this node to the Destination node.
  - Parameters are optional.
  - **receive(MessageType[, Parameters])**
  - A message of type MessageType is received by this node.
  - Parameters are optional.

Sending and Receiving Messages

node 5

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

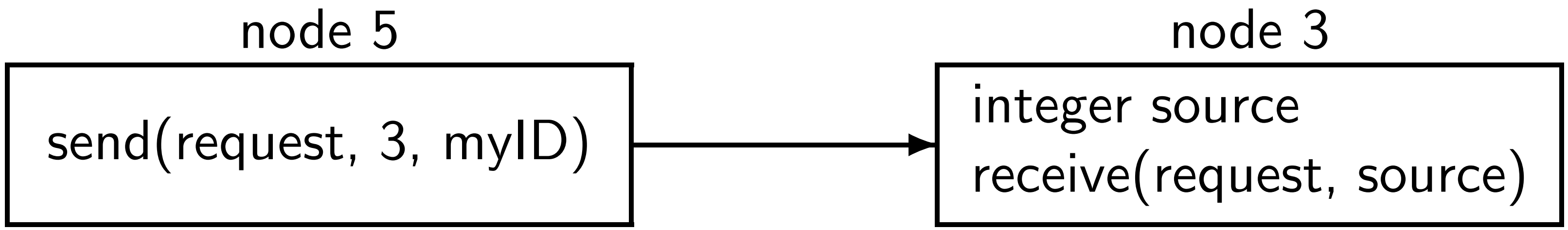
→

node 3

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

- The algorithm of the sending node must pass the destination ID to the underlying communications protocol.
- Neither the ID of the source node nor the ID of the destination node need to be sent within the message.
- The process that executes the receive statement blocks until a message of the proper type is received.
- Then the values are copied into the variable parameters and the process is awakened.

Sending a Message and Expecting a Reply



- If the source node wishes to make its ID known to the destination node, it must include myID as an explicit parameter.

Implementations

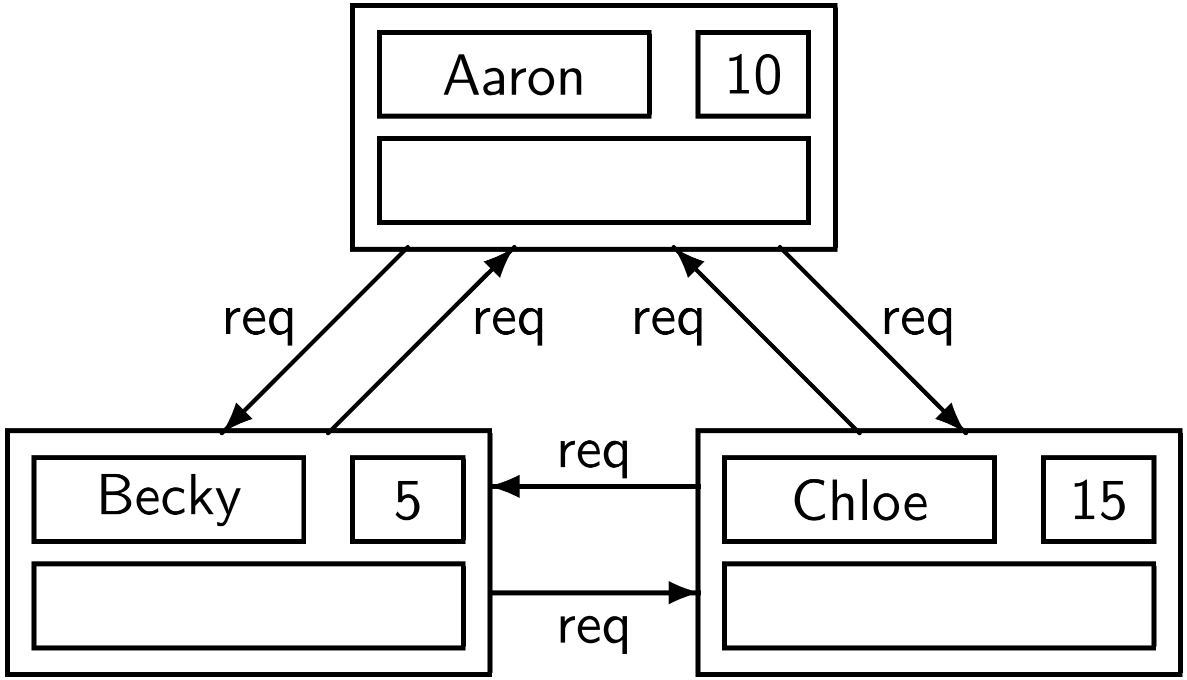
- **Parallel Virtual Machine (PVM)**
  - Software system that presents the programmer with a virtual distributed machine.
  - Programmers can freely assign processes to nodes.
  - The architecture of the virtual machine can be dynamically changed by any node in the system (so that PVM can be used to implement fault-tolerant systems).
  - Portable over a wide range of architectures.
- **Message Passing Interface (MPI)**
  - Designed to emphasise performance by letting each implementation use the most efficient constructs on the target computer.
  - Significantly different from PVM (no details here).

Distributed Mutual Exclusion

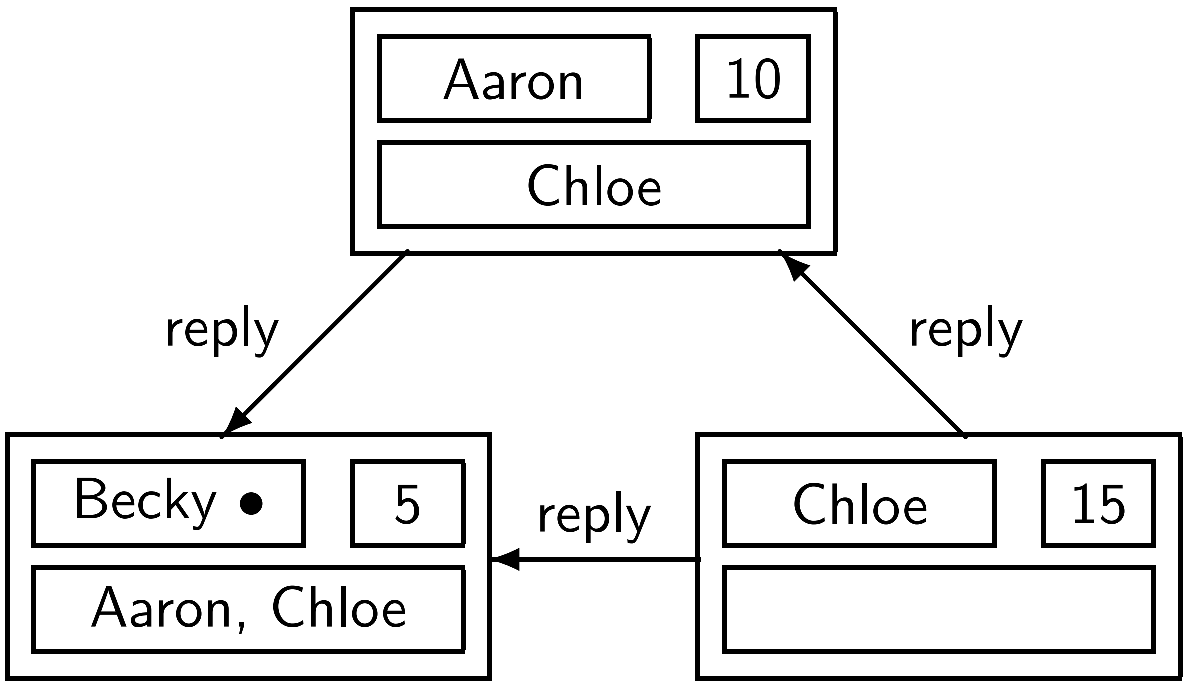
- Glenn Ricart and Ashok K. Agrawala proposed an algorithm based on ticket numbers as in the bakery algorithm.
- Nodes choose ticket numbers and compare them.
- The lowest number is allowed to enter the critical section.
- Since numbers cannot be directly compared in a distributed system, they must be sent in messages.

Algorithm 10.1: Ricart-Agrawala algorithm (outline)	
	integer myNum ← 0 set of node IDs deferred ← empty set
main	
p1:	non-critical section
p2:	myNum ← chooseNumber
p3:	for all *other* nodes N
p4:	send(request, N, myID, myNum)
p5:	await reply's 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)

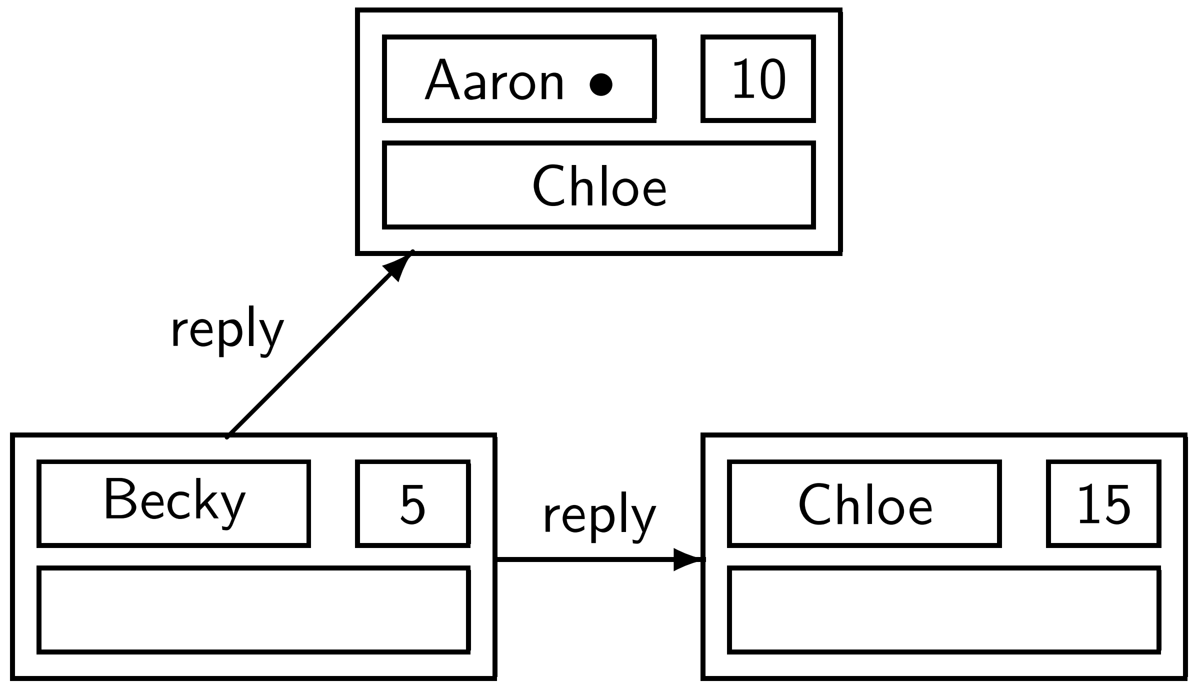


RA Algorithm (2)

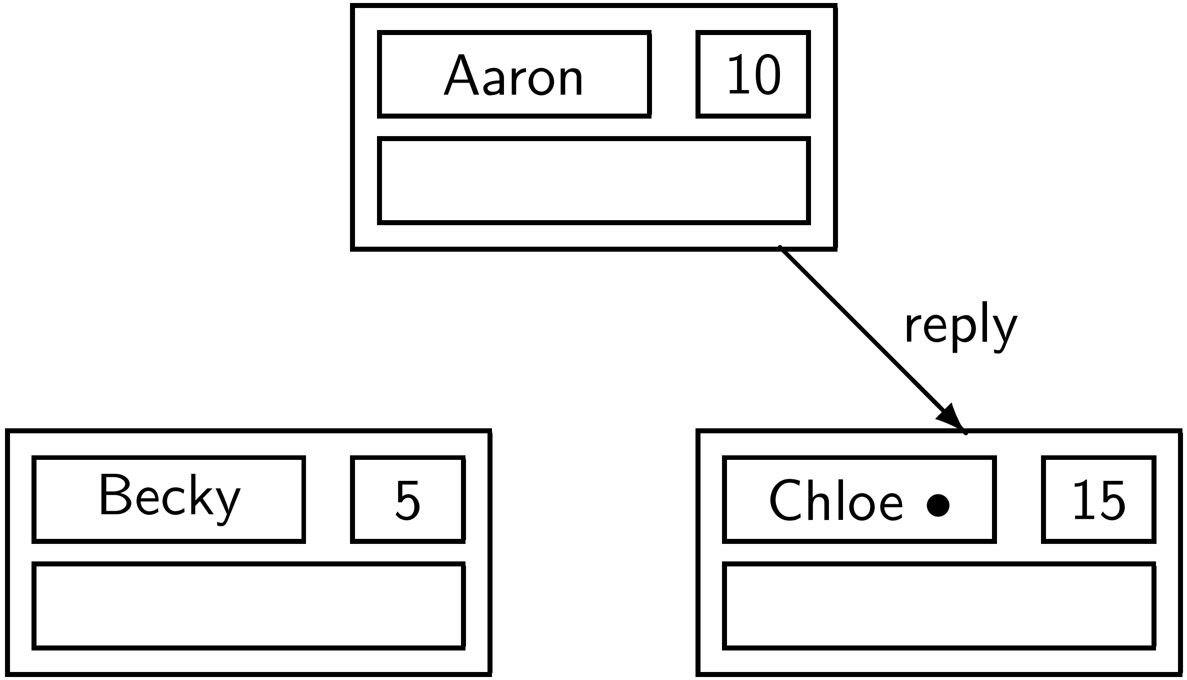


Virtual Queue in the RA Algorithm

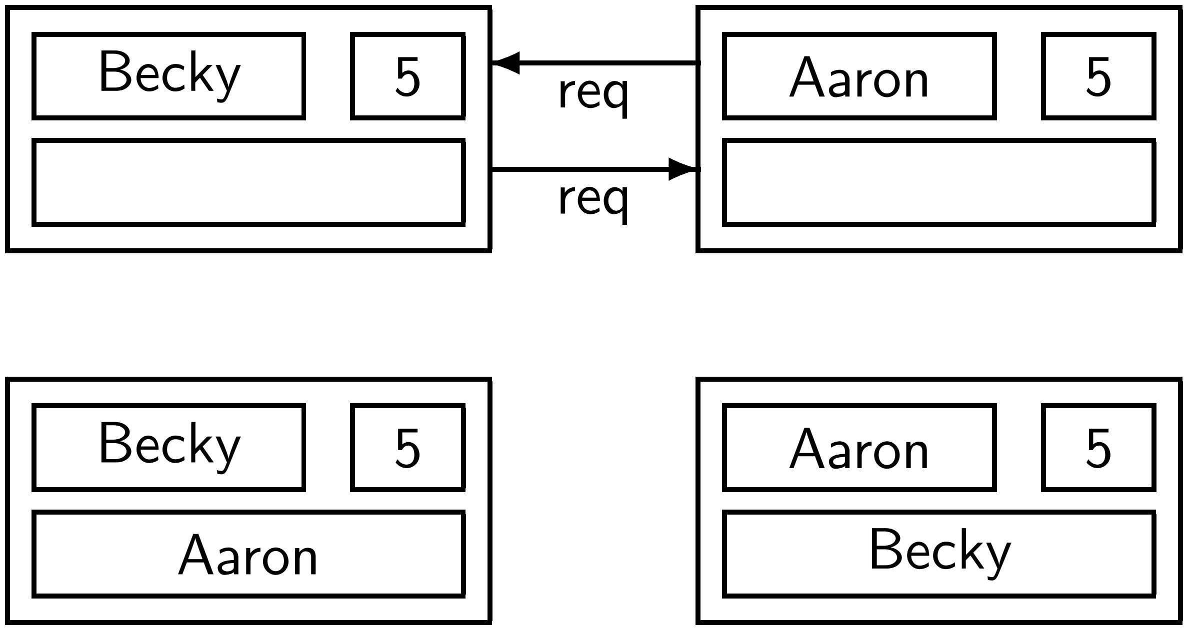
RA Algorithm (3)



RA Algorithm (4)



Equal Ticket Numbers

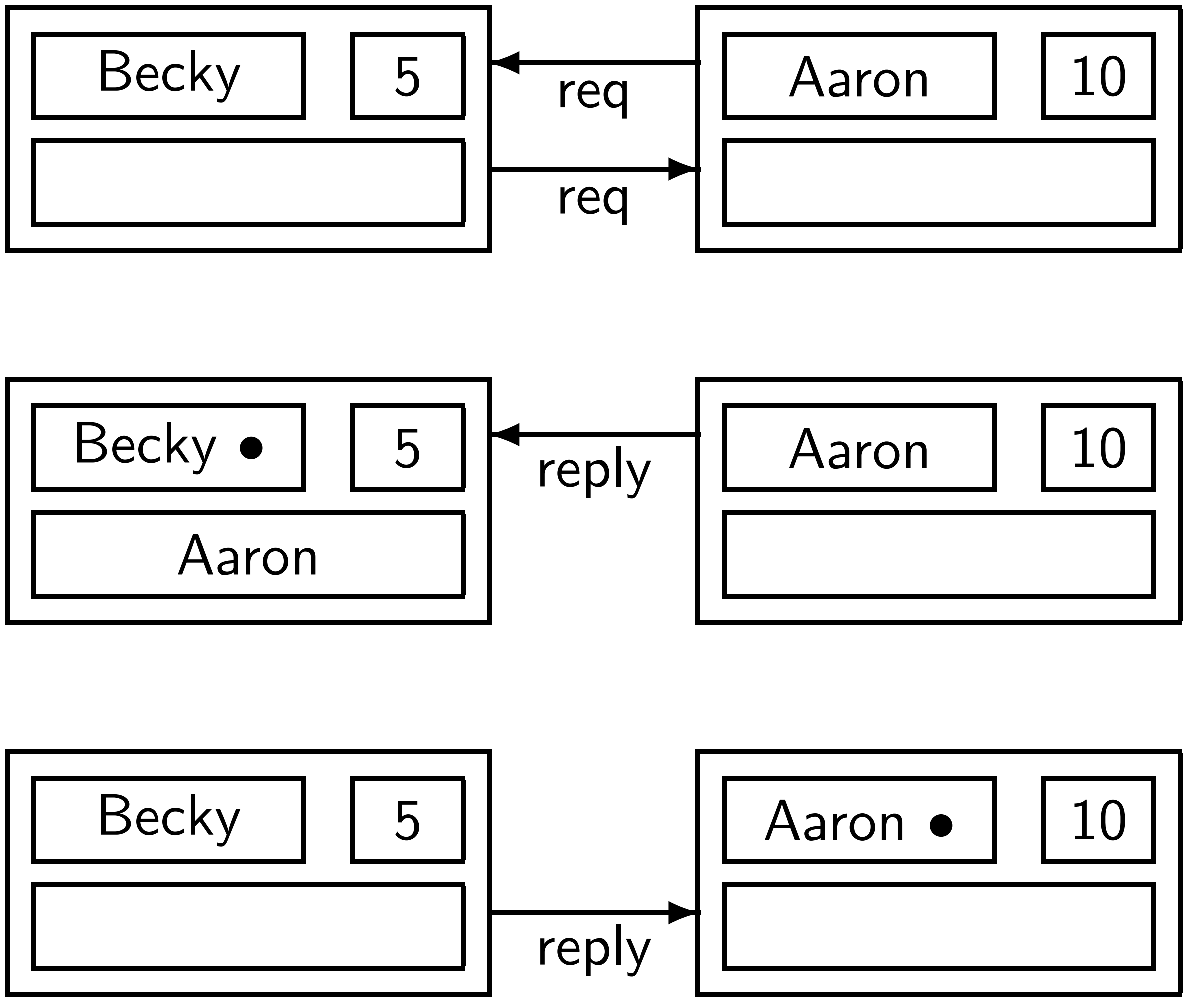


- IDs will be used to break ties if processes have chosen the same number:

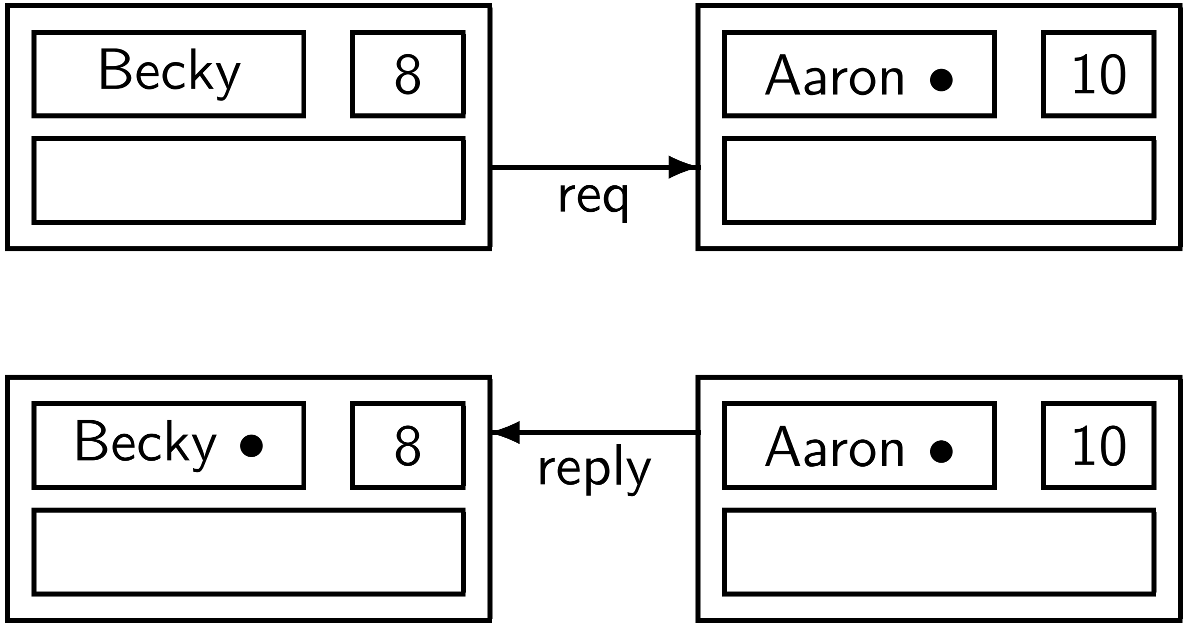
  (requestedNum < myNum) or
  ((requestedNum = myNum) and (source < myID))

- The notation "requestedNum ≪ myNum" is used as an abbreviation for this test.

Choosing Ticket Numbers (1)

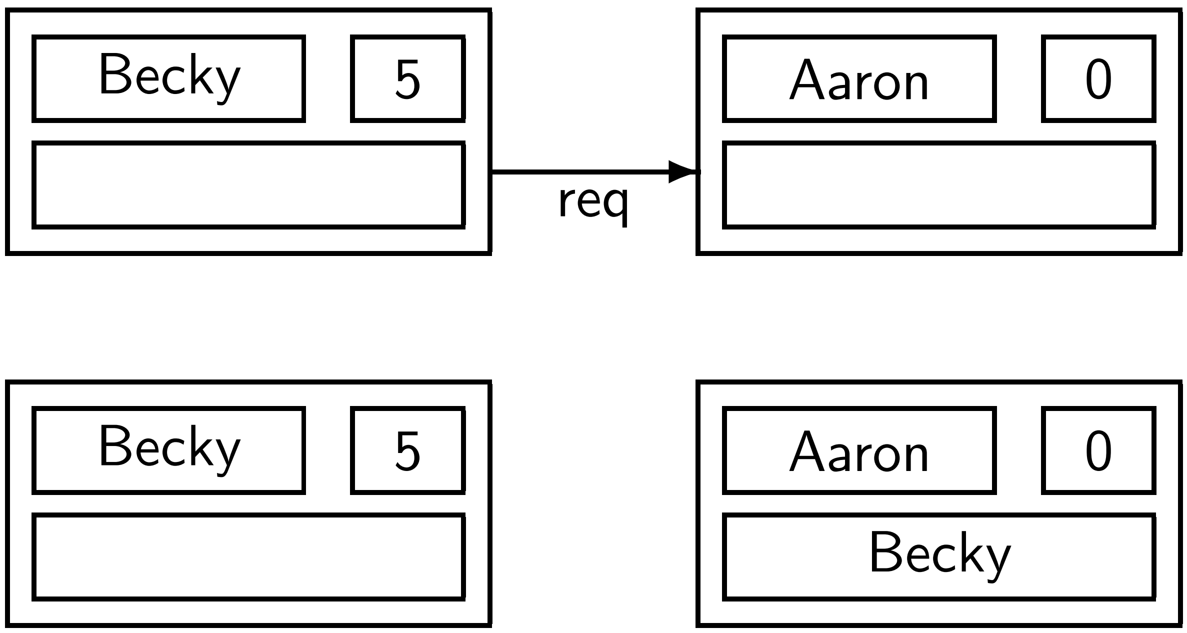


Choosing Ticket Numbers (2)



- Mutual exclusion is not guaranteed if numbers are not chosen in monotonic order.
- To prevent this from happening, each node stores the highest number received in any request message.
- A new ticket gets a number that is higher than the highest number received.

Quiescent Nodes



- Receive will only send a reply if myNum is greater that requestedNum.
- If a node remains inactive for a long period, other nodes cannot enter the critical section because of the ever-increasing ticket numbers.
- To solve this problem, a flag requestCS is added which the Main process sets before choosing a ticket number and resets upon exit from its critical section.
- If the flag is not set, the Receive process will immediately send a reply.

Algorithm 10.2: Ricart-Agrawala algorithm

```
                    integer myNum ← 0
                    set of node IDs deferred ← empty set
                    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 reply's 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 10.2: 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 ≪ myNum
p4:         send(reply, source, myID)
p5:      else add source to deferred
```

- The algorithm guarantees mutual exclusion.
- It is free from starvation and therefore free from deadlock.

Token-Passing Algorithms

- A permission-based algorithm can be inefficient if there are a large number of nodes.
- Furthermore, it does not show improved performance in the absence of contention.
- In token-based algorithms, permission to enter the critical section is associated with the possession of a token.
- Mutual exclusion is trivially satisfied by a token-based algorithm.
- Furthermore, it is efficient, because only one message is needed to transfer the token.
- The challenge is to construct an algorithm for passing the token that ensures freedom from deadlock and starvation.

Algorithm 10.3: Ricart-Agrawala token-passing algorithm
boolean haveToken $\leftarrow$ true in node 0, false in others integer array[NODES] requested $\leftarrow$ [0,. . . ,0] integer array[NODES] granted $\leftarrow$ [0,. . . ,0] integer myNum $\leftarrow$ 0 boolean inCS $\leftarrow$ false
sendToken if exists N such that requested[N] $>$ granted[N] for some such N send(token, N, granted) haveToken $\leftarrow$ false

- The array granted holds ticket numbers held by each node last time it was granted permission to enter the critical section.
- The array requested holds the ticket numbers sent with the last request messages.

Algorithm 10.3: 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 10.3: 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
```

- The algorithm guarantees mutual exclusion.
- It is free from deadlock but not from starvation.

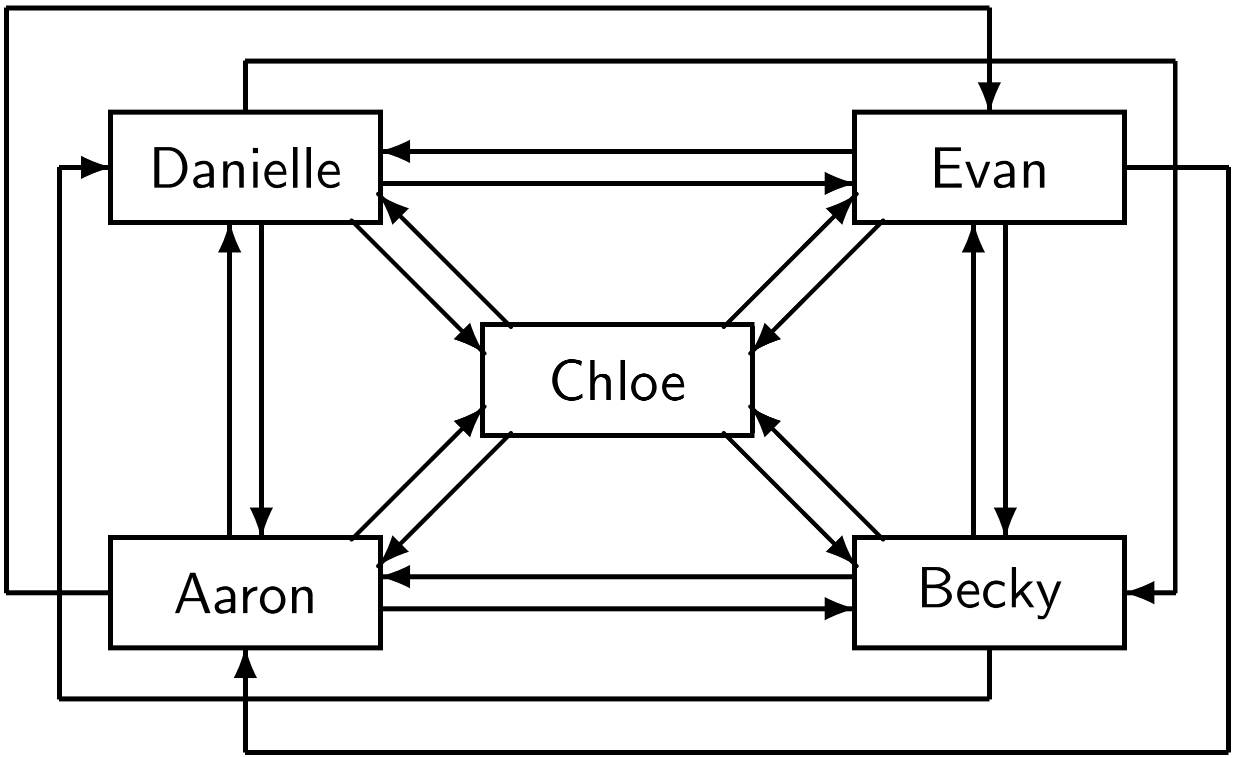
Data Structures for RA Token-Passing Algorithm

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

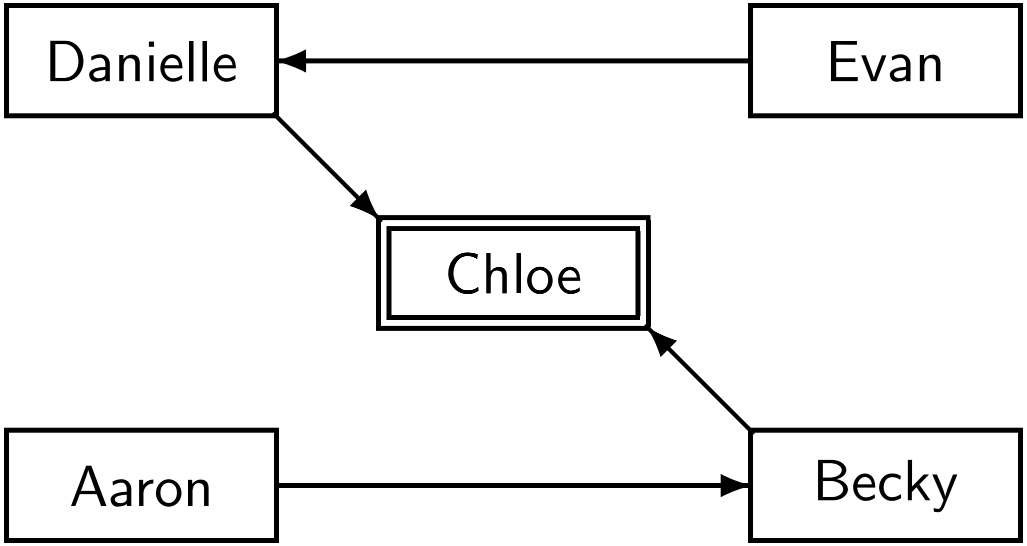
Tokens in Virtual Trees

- The RA token-passing algorithm carries the queue of waiting processes along with the token message, which results in long messages.
- The Neilsen-Mizuno algorithm avoids this problem by passing a small token in a set of virtual trees.
- The virtual trees are constructed implicitly by the algorithm.
- The initial tree is an arbitrary spanning tree with directed edges pointing to the root.
- The node at the root of the tree possesses the token.

Distributed System for Neilsen-Mizuno Algorithm



Spanning Tree in Neilsen-Mizuno Algorithm



- Chloe possesses the token and can enter her critical section repeatedly until she receives a request message.

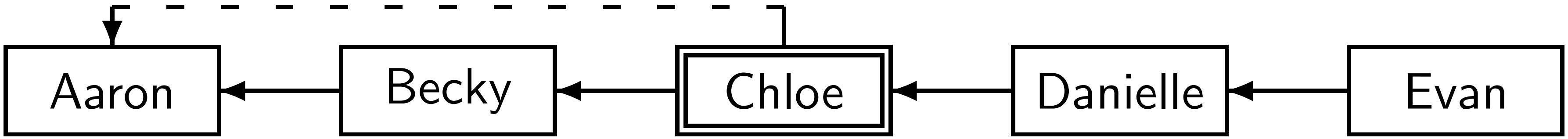
Neilsen-Mizuno Algorithm (1)

- If Aaron wishes to enter his critical section, he sends to his parent, Becky, the message (request, Aaron, Aaron).
- The first parameter is the ID of the sender, while the second one is the ID of the originator (which are the same in this case).
- After sending the message, Aaron zeroes out his parent field, indicating that he is the root of a new tree.

Neilsen-Mizuno Algorithm (2)

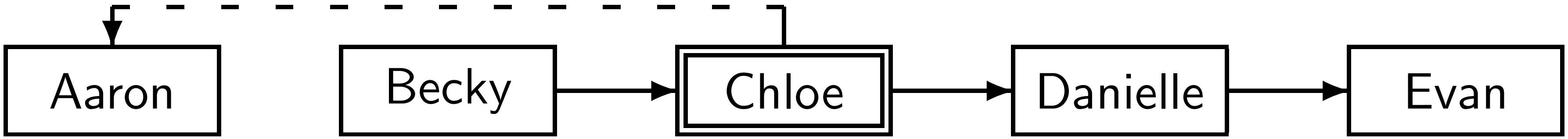
- Becky relays the request to the root by sending the message (request, Becky, Aaron).
- She then changes her parent field to the sender of the message, Aaron.

Neilsen-Mizuno Algorithm (3)



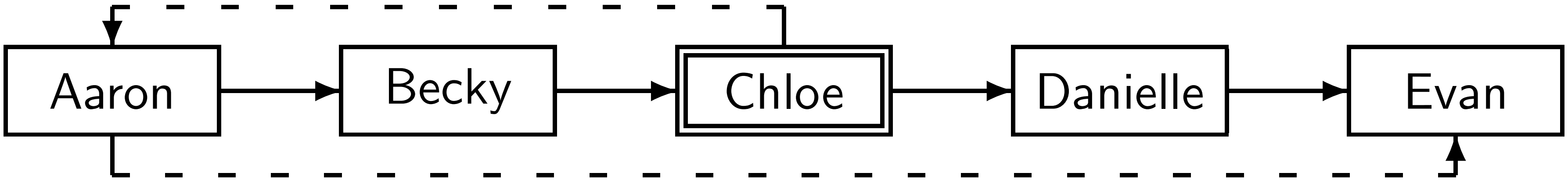
- The node receiving Becky's message, Chloe, possesses the token.
- If Chloe is in her critical section, she sets a field deferred to the value of the originator parameter in the message.
- She also sets her parent field to the sender of the message so that she can relay other messages.

Neilsen-Mizuno Algorithm (4)



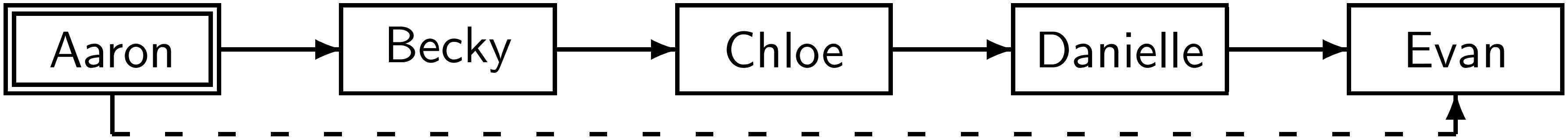
- Suppose that Evan concurrently originates a request to enter his critical section and that his request is received by Chloe after Aaron's request.
- Since Chloe is no longer the root node, she simply relays the message.
- She also sets her parent field to Danielle.
- The chain of relays continues until the message (request, Becky, Evan) arrives at the root, Aaron.

Neilsen-Mizuno Algorithm (5)



- Aaron is a root node without the token, so he knows that he appears in the deferred field of some other node.
- Therefore Aaron places the originator of the message in his deferred field.
- The deferred fields implicitly define a queue of processes waiting to enter their critical sections.
- Aaron also sets his parent field to the sender, Becky.

Neilsen-Mizuno Algorithm (6)



- When Chloe finally leaves her critical section, she sends a token message to the node listed in her deferred field, Aaron.
- This enables Aaron to enter his critical section.

Neilsen-Mizuno Algorithm (7)

- When Aaron leaves his critical section, he sends a token message to the node listed in his deferred field, Evan.
- This enables Evan to enter his critical section.

Algorithm 10.4: 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 10.4: 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
```