

Distributed Operating Systems

8th Chapter
Mutual Exclusion

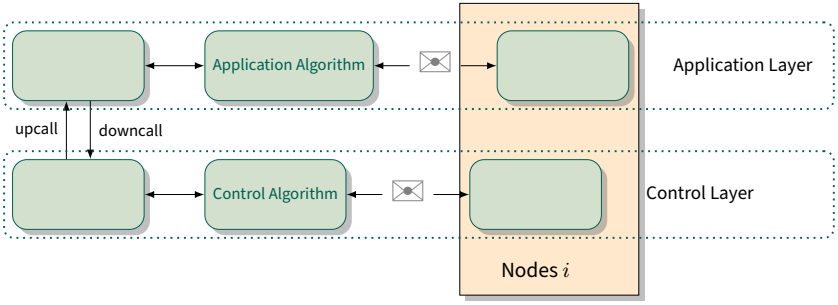
Matthias Werner
Chair Operating Systems

8.1 Fundamentals

- Coordination of exclusive resource access
  - Program section that implements access is called critical section
  - Resource examples: file, event queue, printer
- **Assumption**: A process owning the access right, gives the access right up again after a finite amount of time
- Usually: **At maximum one** process is allowed to access a resource
  - Sometimes generalization: at maximum n processes are allowed to access resources simultaneously; potentially different types of accessors/accesses ➡ reader-writer-problem

Control Algorithms

- **General**: Mutual Exclusion i.a. coordination is necessary in distributed and non-distributed operating systems
  - Problem in distributed OS more difficult
- What, if programs (algorithms) are distributed themselves?
- ➡ Algorithms for determination/generation of the state of distributed algorithms
- Differentiation: **application algorithm** and **control algorithm**



Requirements on an Implementation

- Control algorithms need to exhibit specific properties
- **Safety**: something irreversible bad shall never happen
  - Here: at each point in time maximum one process may be granted access
- **Liveness**: something that shall happen, will finally happen
  - Here: if there is at least one applicant, access has to be granted to an applicant after a finite amount of time
- Algorithms usually have to fulfill safety **and** liveness
- If one demands **only one** of both, usually a trivial solution is possible
  - **Example**: safety only ➡ access is always denied
    liveness only ➡ access is always granted

Requirements on an Implementation (cont.)

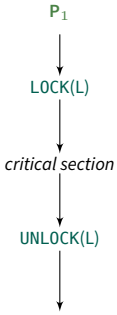
- Usually required additionally: **Fairness**
  - **No starvation**: if a process requires access, access has to be granted after a finite amount of time
  - **Stronger fairness requirement**: granting (somehow) takes the order of requests into account

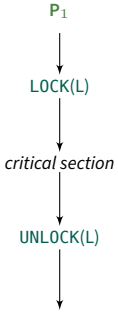
Performance Criteria

- To evaluate the **performance** of a mechanism for mutual exclusion diverse criteria may be adduced
  - **Number** n_m of sent messages per CS (message complexity) ($n_m \to \min$)
  - **Vote delay** d per CS ($d \to \min$)
  - **Response time** t_r, meaning the time span from requesting the CS till leaving ($t_r \to \min$)
  - **Throughput** T_P, meaning the number of iterations through a CS per time unit ($T_P \to \max$)
- Following, focus on message complexity and vote delay

Repetition: Mutual Exclusion in Non-distributed Systems

- In centralized systems, there is an implementation for mutual exclusion of kernel operations of the OS
- Implementation is in the end based on a shared variable



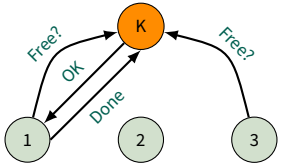


Repetition: Mutual Exclusion in Non-distributed Systems (cont.)

- Diverse mechanisms are known (i.a. from Lecture Operating Sytems I):
  - Spin Lock
  - Process Lock
  - Semaphor
  - Monitor
- Mechanisms are based on atomic access to shared memory (**atomic** testing and setting of memory cells)
- Not given in distributed systems!
- How can mutual exclusion be implemented in distributed systems?

Centralized Solution for Distributed Systems

- Reduction to local case:
  a process is coordinator regarding to a resource (e.g. through a selection algorithm, see later chapter)
- All requests and releases are communicated to the coordinator
- Coordinator assigns allocations
- Implementation is simple
- 3 messages per access



Distributed Coordination

- Centralized solutions are simple to implement, but have disadvantages too
  - “Single point of failure”
  - Bottleneck
  - Delay: at least double message delivery time
- Hence, we need to think about distributed solutions
- **Assumption**: in distributed case, operations `LOCK` and `UNLOCK` exist
- Additional auxiliary processes may be necessary

8.2 Broadcast-based Algorithms

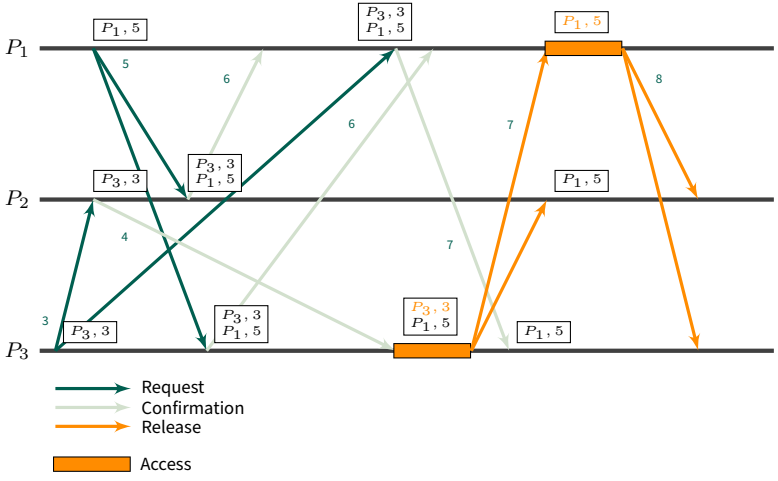
Broadcast Algorithm of LAMPORT, 1978

- **Assumptions**:
  - Lossless FIFO communication channels
  - All messages have unique logical time stamps ➡ see chapters “Group Communication” and “Logical Clocks”
- **Basic Idea**:
  - Each process manages a message queue ordered by time stamps
  - Requests and releases are sent **to all** via broadcast

Broadcast Algorithm of LAMPORT, 1978 (cont.)

- **Sent request**: Enqueue request with time stamp in own queue and sent to all
- **Receive request**: Enqueue request in own queue; sent request confirmation to requesting processes
- **Sent release**: Remove request from own queue and sent release to all
- **Receive release**: Remove request from own queue
- A process is allowed to access, if
  - own request is **first** request in the queue
  - a message with larger time stamp was already received from **any** other process.

Broadcast Algorithm of LAMPORT, 1978 (cont.)



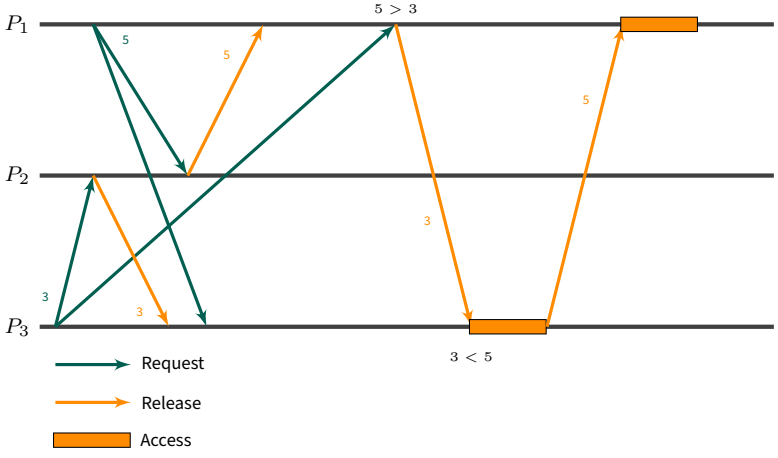
Broadcast Algorithm of LAMPORT, 1978 (cont.)

- Earliest request is **globally unique**, as soon as all other porcesses have received a message with larger logical time stamp
- **Message complexity**:
  - Sent request to $(n-1)$ processes
  - $(n-1)$ processes send a confirmation
  - Sent release to $(n-1)$ processes
  - ➡ $3(n-1)$ messages per access all in all ➡ for boradcast messages same performance as centralized solution, otherwise worse
- **Vote delay**:
  - a single message delay ➡ better than centralized solution

Improved Broadcast Algorithm

- Ricart and Agrawala, 1981
- Does not require FIFO-channels
- Needs only $2(n-1)$ messages per access
- **Basic Idea**:
  - Sent **request** (with time stamp) to all other processes $(n-1)$
  - Access, as soon as $n-1$ agreements **for this request** have been received
  - ➡ Each request must have a unique ID
- **At arrival of a request**:
  - Sent agreement immediately, if
    - there is no own request
      or
    - Sender has "older rights" (noticeable via logical time stamp)
    - If time stamps are equal, decision via node ID
  - Else, sent agreement **after termination** of own access

Improved Broadcast Algorithm (cont.)



- Numbers of agreement messages show the time stamp of the corresponding request, not the time stamp of the message itself

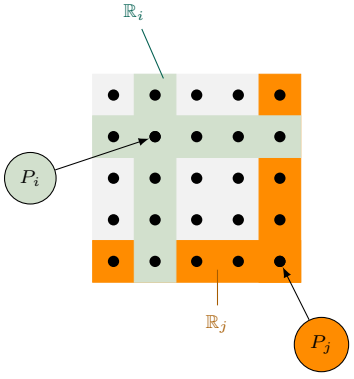
8.3 Quorum-based Algorithms

Better Algorithms?

- Is a solution possible that
  - works with **less messages** per access
    and
  - nevertheless provides load-balancing over all processes?
- Is there a solution that
  - works without participation of all nodes in each coordination
    and
  - nevertheless provides load-balancing over all processes?

Process-grid Algorithm

- MAEKAWA, 1985
- The n processes are arranged in a square grid with length $\sqrt{n} \times \sqrt{n}$
- Before, the access, a process P_i has to ask a specific set of processes (its granting set $\mathbb{R}_i$) for permission
- For all pairs of processes P_i and P_j, $\mathbb{R}_i$ and $\mathbb{R}_j$ are arranged in a way that they have at least two processes in common



Process-grid Algorithm (cont.)

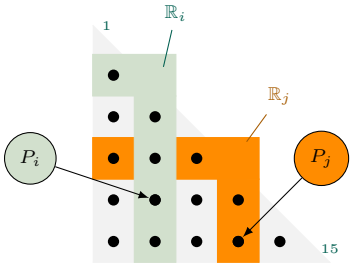
- Granting sets have [1] the cardinality $2\sqrt{n} - 1$
- **Message complexity**:
  - Sent request to $2(\sqrt{n} - 1)$ processes
  - $2(\sqrt{n} - 1)$ processes send agreement
  - Send release to $2(\sqrt{n} - 1)$ processes
  - ➡ all in all, $6(\sqrt{n} - 1$ messages per access)
- **Problem**: Deadlocks may occur
  - Avoidable through the introduction of further message types
  - Increases message complexity by a constant factor only

Is there another arrangement of processes for which the cardinality of the granting sets is smaller?

[1] including the applicant

Triangular Arrangement

- In a square grid, at least two different granting sets have each two processes in common
- However, a **single** process in common would be sufficient!
- **Solution**:
  - Processes are arranged in a triangle
  - Granting sets have roughly the cardinality $\sqrt{2n}$
- **Problem**: Agreement of some processes is more often necessary than the agreement of other processes!
  - Process 15 is in **one** granting set only
  - Process 1 is in **nine** granting sets

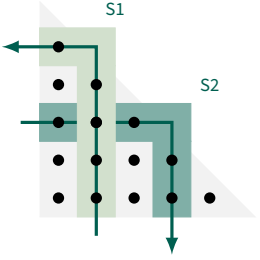


Formation of granting set

Same column and row one above as topmost node of column

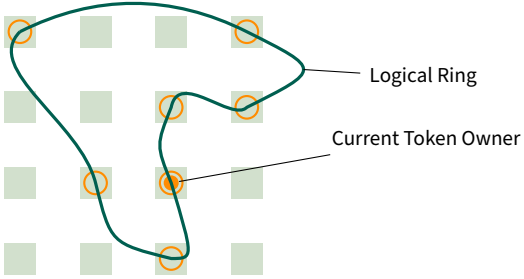
Solution for Load-balancing

- **Approach**: **two different** schemes
  - S1: Same column and row one above topmost node of column (to the top and to the left)
  - S2: Same row and column one futher right than rightmost node of row (to the right and to the bottom)
- **Properties**
  - Each granting set overlaps with each granting set of the **same** scheme
  - Each granting set of **a** scheme overlaps with each granting set of **the other** scheme
  - All processes occur **equally often** in S1∪S2
- Alternating (or random) use of both schemes ➡ Load-balancing



8.4 Token-based Algorithms

- **Topology**: the nodes participating in the mutual exclusion form a specific topology, e.g. a logical ring
  - Alternatively, a real topology is used
- There is exactly one **access permission** that is passed along the topology



A simple token ring solution

- Le Lann, 1977
- Processes are placed in a logical ring
- Each resource has its own token
- Access is controlled by a circulating token
- Processes wait for access till they get the token
- If a process finished its access it passes the token on
- A process without the intention to access a resource simply passes the token on

A simple token ring solution (cont.)

- **Advantages**
  - Simple, correct and fair algorithm
  - No deadlocks
  - No starvation
  - Priorities are possible
- **Problems**
  - Token circulates all the time, maybe unnecessarily
  - ➡ Number of messages per request is not bounded
  - Long waiting time for big number of processes

Lift-Algorithm

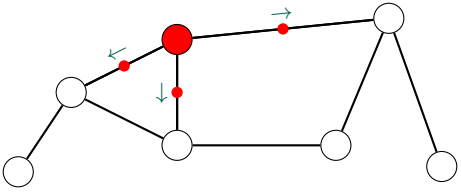
- **Alternative Idea**: using a **spanning tree**
- **Advantage**: no concurrent paths
- **Problem**: How does one obtain a spanning tree?
  ➡ **Echo-Algorithm**

Excursus: Echo-Algorithm

- The Echo-Algorithm is used for the **construction of spanning trees**
- It can be used for arbitrary connected communication topologies
- Used as a sub algorithm in algorithms for e.g.
  - mutual exclusion (here 😊)
  - Termination Problem
  - Selection Problem

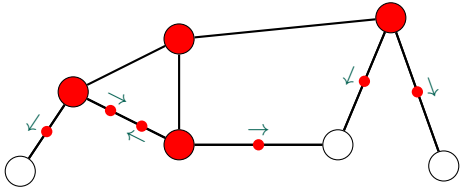
Excursus: Echo-Algorithm (cont.)

- State of a node is described via a color
- Initially all nodes are **white**
- The unambiguous initiator is colored **red** and sends a red messages (**Explorer**) to all its neighbors



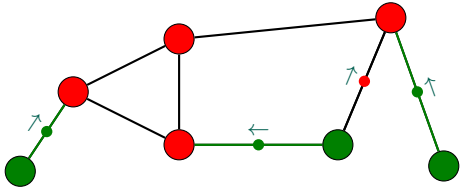
Excursus: Echo-Algorithm (cont.)

- A white node that receives an explorer is colored red, memorizes this “first” edge (**activating edge**) and sends explorers to all its neighbors
- If an explorer is received from a red node the explorer is not passed on (its “swallowed”)



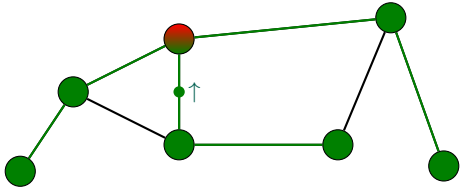
Excursus: Echo-Algorithm (cont.)

- A red node that received over **all** adjacent edges either an echo or an explorer is colored **green** and sends a green **echo** via its activating edge (that is also colored green)
- If a **Leave** receives an explorer it immediately replies an echo



Excursus: Echo-Algorithm (cont.)

- Step by step all nodes and some edges are colored green
- The algorithm terminates if the initiator received an echo or an explorer over all adjacent edges

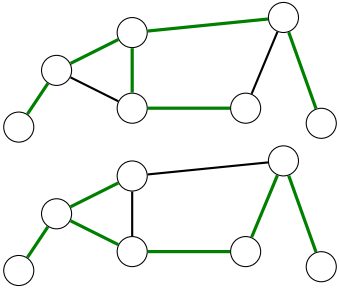


Echo-Algorithm – Properties

- Over each edge exactly two messages are passed
  - Either an explorer and an echo (in the opposite direction) or two explorer (in different directions)
- Parallel traversal of an connected undirected graph with $2e$ messages
- The echo-algorithm is an **Wave Algorithm**
  - **There wave**: nodes are colored red
    ➡ distribution of information to all nodes
  - **Back wave**: nodes are colored green
    ➡ collection of information from all nodes

Echo-Algorithm – Properties (cont.)

- Echo-edges constitute a **spanning tree**
- Depending on the message delay the spanning tree looks different because fast edges are preferred over slow ones



Improvements of the Echo-Algorithm?

- **Idea**: avoid edges for which it is known that other explorers already visited them
- With each explorer a set of taboo-nodes $\mathbf{Z}$ is sent and received
- The initiator sends $\mathbf{Z} = <$ neighbors of the initiator $> \cup <$ initiator $>$ as the taboo-set
- Explorers are just sent to neighbors $\mathbf{Y}$ who are not in $\mathbf{Z}$
  - the taboo-set $\mathbf{Z}' = \mathbf{Z} \cup \mathbf{Y}$ is attached
- **Advantage**: Reduction of the number of messages
- **Disadvantage**:
  - Message length of $\mathcal{O}(n)$
  - The identity of all neighbors has to be known

Lift-Algorithm

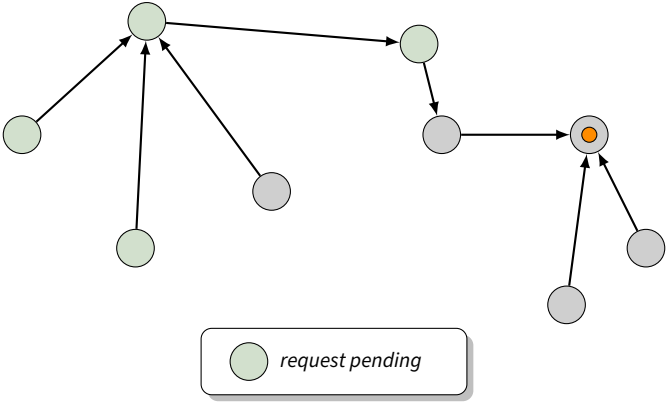
Back to the lift-algorithm: using spanning trees

- Edges of the spanning tree have a **state** (direction)
  - They each can point in one of two directions
- Selectively forwarding of access request in the direction of the token (instead of sending to all processes)
  - Token moves **in the opposite direction of the edges** and **reverses** direction with each edge it passes
  - A process that wants the token sends a corresponding request over all its **outgoing** edges
- If a process received a request it sends a request after the token **one time only**
  - Further requests are “memorized”

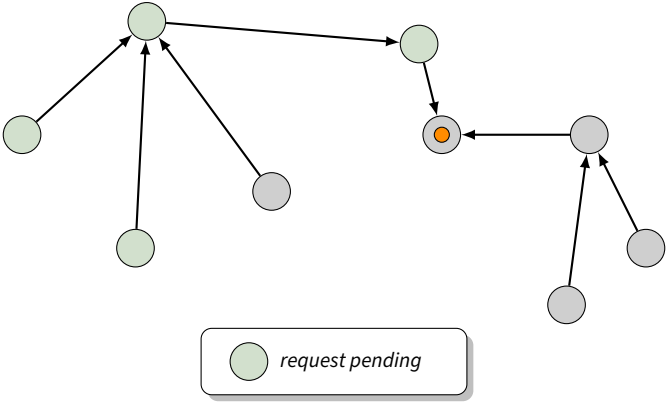
Lift-Algorithm (cont.)

- If a process receives the token ...
  - it is allowed to access if it was its request
  - it forwards it in **one** of the requesting directions
  - if there are other (memorized) request form other directions it sends a request (**after**) the token
- To ensure fairness a process is not allowed to serve requests from the same direction **arbitrarily often**

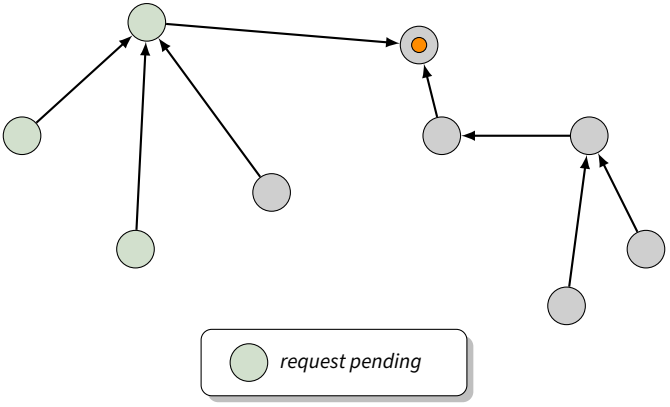
Lift-Algorithm (cont.)



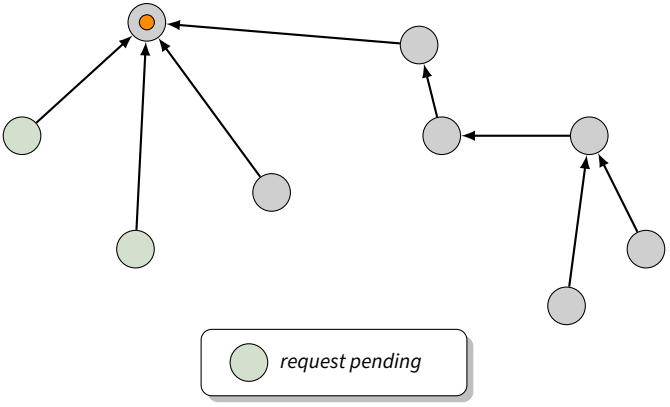
Lift-Algorithm (cont.)



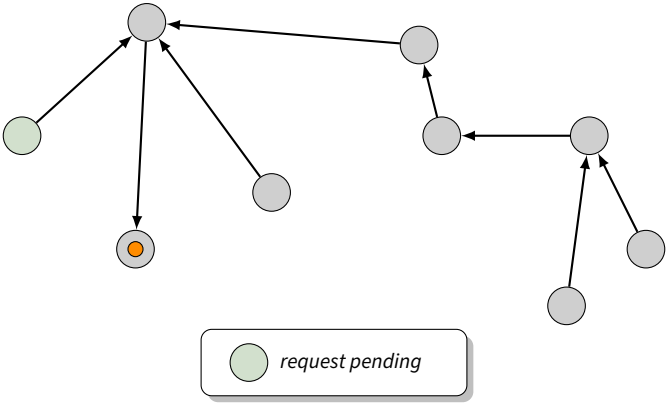
Lift-Algorithm (cont.)



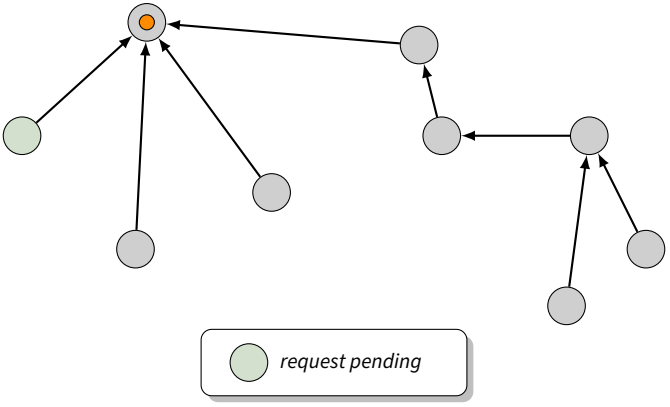
Lift-Algorithm (cont.)



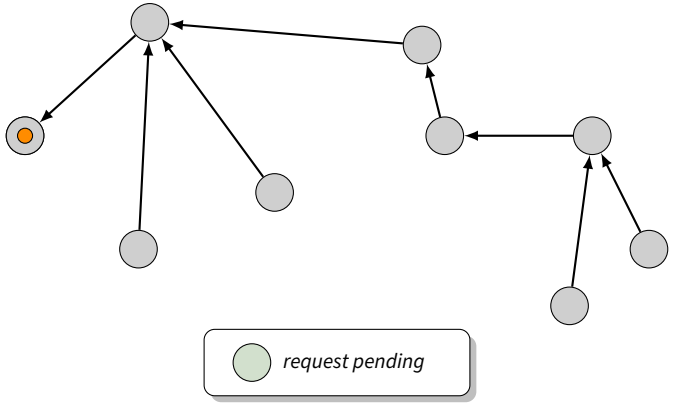
Lift-Algorithm (cont.)



Lift-Algorithm (cont.)



Lift-Algorithm (cont.)



Lift-Algorithm (cont.)

- **Invariant**: There is a directed path from each process to the token
- In a balanced k-tree the maximum length between arbitrary processes is $\mathcal{O}(\log_k n)$
- Hence there are just $\mathcal{O}(\log_k n)$ messages per access needed
- **Initial state**: Winner of a selection (see later lectures) gets the token and starts the echo-algorithm to generate a spanning tree where the edges are directed towards him
- This procedure can be generalized to arbitrary connected topologies

Literature

[RiAg81] Ricart, G. and A. K. Agrawala: "An Optimal Algorithm for Mutual Exclusion in Computer Networks". *Communications of the ACM*, 24(1)1981, 9–17

[Mae85] Maekawa, M.: "$\sqrt{n}$ Algorithm for Mutual Exclusion in Decentralized Systems". *ACM Transactions on Computer Systems*, 3(2)1985, 145–159

[LuWo97] Luk, W. S. and T. T. Wong: "Two New Quorum Based Algorithms for Distributed Mutual Exclusion", in *17th International Conference on Distributed Computing Systems (ICDCS '97)*, 100–107

[SuKa85] Suzuki, I. and T. Kasami: "A distributed mutual exclusion algorithm". *ACM Transactions on Computer Systems*, 3(4)1985, 344–349