Distributed Algorithms 2016/17

Mutual Exclusion

Odej Kao | Complex and Distributed IT Systems

With material of R. Karnapke | KBS

Overview

Problem of mutual exclusion

Algorithm with central coordinator

Broadcast-based algorithms

Quorum-based algorithms

Token-based algorithms

Comparison of algorithms

Mutual Exclusion

Coordination of the exclusive access on resources

– Examples for resources: file, printer

Often, only 1 process shall access the resource

Sometimes instead maximal *n* processes may access at the same time ($n > 1$)

Assumption: If a process has the right to access, he releases it after finite time voluntarily

Default for the lecture

Requirements for a Realization

Safety: Something bad that cannot be undone shall never happen

- Here: At no point in time must an access be allowed for more than one process

Liveness: Something that should happen eventually happens

- Here: If there is at least one applicant, the access has to be allowed to one of the applicants after finite time

Algorithms must fulfill safety *and* liveness; often, a trivial solution is possible for only *one* of the two

Requirements for a Realization

Often required additionally besides Safety and Liveness: **Fairness**

- No starvation: If a process desires access, the access has to be allowed after finite time
- Stronger fairness requirements: The allowance of access takes the order of access requests into account

Solutions for Centralized Systems

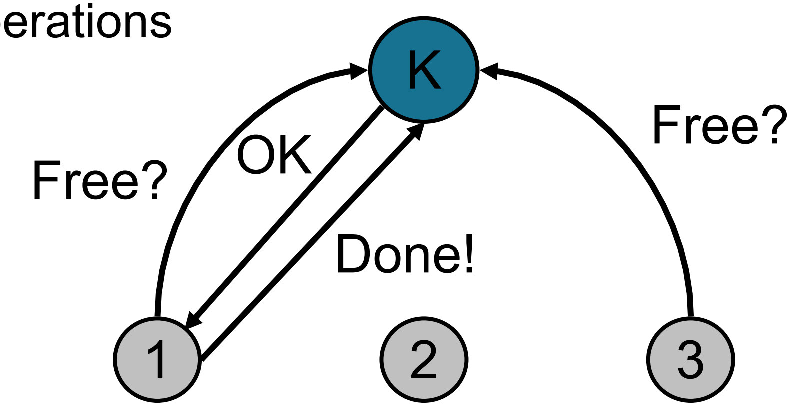
- Examples for used mechanisms to achieve mutual exclusion
  - Busy Waiting
  - Semaphores
  - Monitors
- Those mechanisms are based on the fact that processes can atomically access a common physically memory (atomic testing and setting of a memory cell)
- Not given in distributed systems!
- How can mutual exclusion be realized in distributed systems?

Algorithm with Central Coordinator

Centralized Solution for Distributed Systems

- A process is assigned as coordinator in reference to a resource (e.g. by election)
- The coordinator is informed about all requests and releases
- Coordinator grants accesses
- Easy to implement
- 3 messages per access with blocking operations

- Disadvantages
  - Single Point of Failure
  - Asymmetrical load distribution



Broadcast-Based Algorithms

Broadcast-Algorithm (Lamport, 1978)

Assumptions

- Lossless FIFO-Communication channels
- All messages bear unique logical time stamps

Basic Idea

- Each process manages a message queue ordered according to time stamps
- Requests and releases are sent to all processes via broadcast

A process must only access if

1. its own request is the first request in its own queue
2. It already received a message from each other process (request confirmation or request) with a larger time stamp

Broadcast-Algorithm

Issue access request
- Insert request into own queue
- Send it to all other processes

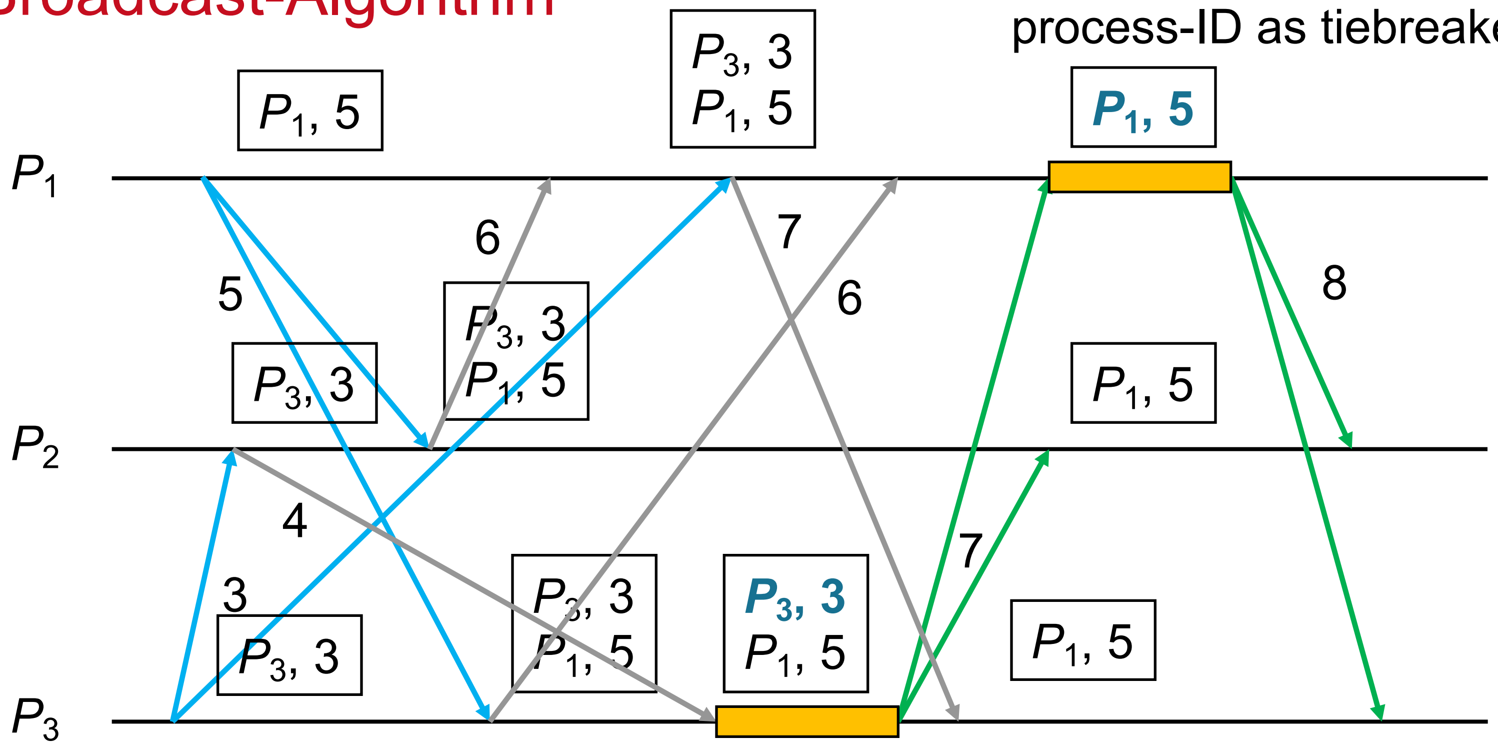
Receive access request
- Insert request into own queue (ordered by timestamp!)
- Send request confirmation to requesting process

Send release after access
- Remove (own) request from own queue
- Send release to all other processes

Received release
- Remove request from own queue

Broadcast-Algorithm

With the same time stamp: process-ID as tiebreaker



Blue Message: Request
Gray Message: Confirmation
Green Message: Release

Orange time interval: access

Broadcast-Algorithm

Earliest request is globally unique, after all processes have received a message with a larger logical time stamp

Message complexity

- Sending of request to (n – 1) processes
- (n – 1) processes send their confirmation
- Sending of release to (n – 1) processes

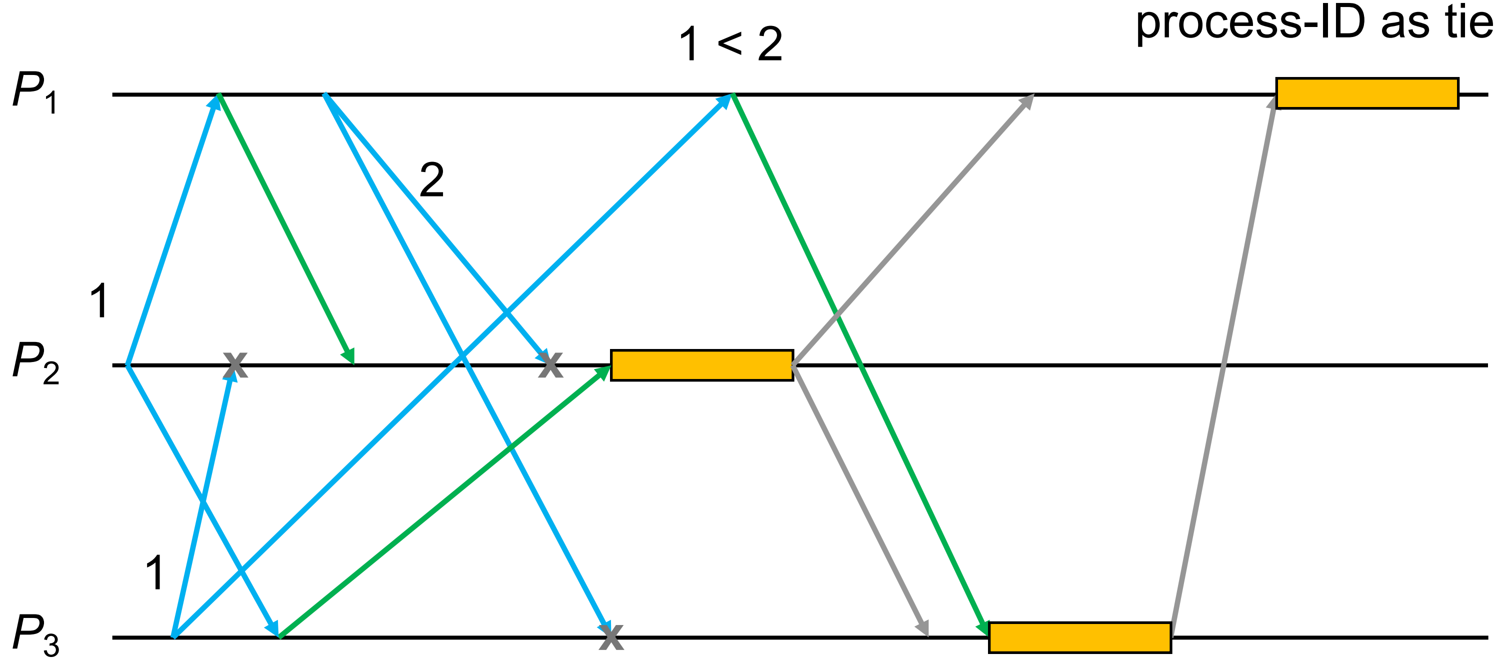
⇒ 3 (n – 1) messages per access altogether

Improvement by Ricart and Agrawala, 1981

- Basic idea: avoid explicit release messages through delayed confirmation → 2 (n – 1) messages per access, no FIFO-channels necessary
- Issue access request
  - For a new request, a sequence number is chosen by the process; the sequence number is by 1 larger than all previously *received* requests
  - Send request to all other n – 1 processes
  - Access after n – 1 confirmations were received
- When a request arrives
  - Send confirmation immediately, if not applied or the sender has „older rights“ (recognizable by sequence number)
    - Same sequence number: Node ID ensures uniqueness
  - Otherwise, confirmation is sent only after the ending of the own access

Improvement by Ricart u. Agrawala, 1981

With the same time stamp process-ID as tiebreaker



Blue Message: Request
Green Message: Immediate Confirmation
Gray Message: Delayed Confirmation

Orange time interval: access

Better Algorithms?

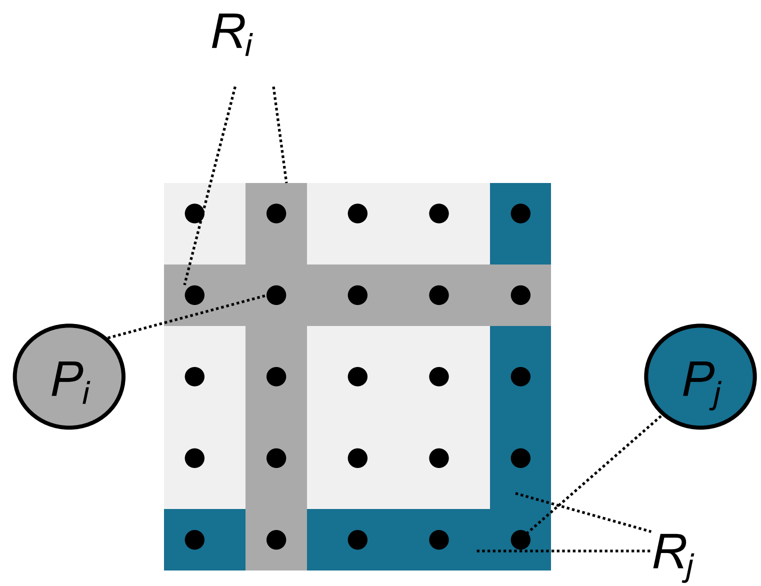
- Is a solution possible that requires less messages per access and that still distributes the load equally between all processes?

- Is there a solution which does not include the involvement of *all* process in *each* coordination and still distributes the load equally between all processes?

Quorum Based Algorithms

Process Mesh-Algorithm (Maekawa, 1985)

- The n processes are arranged in a quadratic mesh with an edge length of $\sqrt{n}$
- A process P_i must ask a certain set of processes (its *granting set* R_i) for allowance before access
- For all pairs of processes P_i and P_j, their R_i and R_j are ordered in such a way that they have at least two processes in common



Same line and column

Process Mesh-Algorithm

Granting sets have the cardinal number $(2\sqrt{n}) - 2$

Message complexity without competing access requests

- Send request to $(2\sqrt{n}) - 2$ processes
- $(2\sqrt{n}) - 2$ processes send confirmation
- Send release to $(2\sqrt{n}) - 2$ processes
- $3[(2\sqrt{n}) - 2]$ messages per access altogether

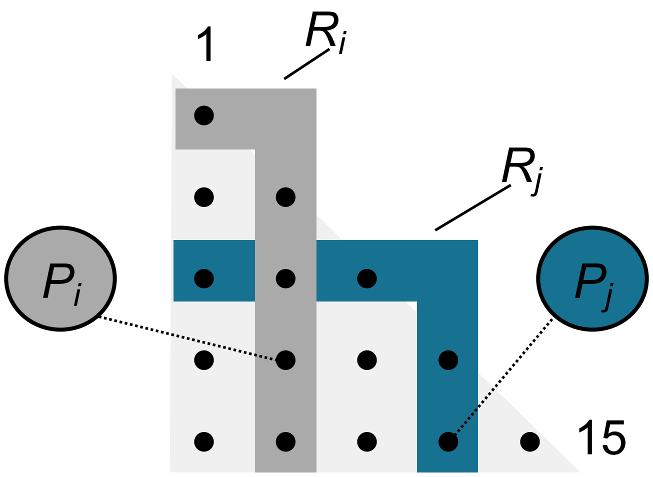
Problem: With competing requests deadlocks may occur

- Avoidable through the introduction of two additional message types
- Increases the number of messages per access on $5[(2\sqrt{n}) - 2]$ in the worst-case

Is there another arrangement of the processes involving a smaller cardinal number of the granting set?

Triangular Arrangement

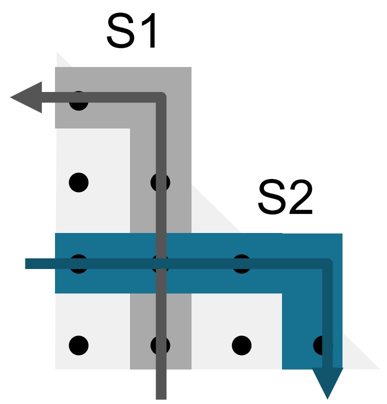
- In a quadratic mesh, two different granting sets have at least two processes in common, but a *single* common process would be sufficient
- Solution: Triangular arrangement of the processes
- Granting sets have a size of about $\sqrt{2}\sqrt{n}$
- Problem: The confirmation of some processes is needed more often than that of other processes!
  - Process 15 only occurs in one granting set
  - Process 1 occurs in 9 granting sets
- Solution for load balancing?



Same column and row one further above than the upper column

Solution for Load Balancing

- The solution is to use two different schemes
  - S1: Same column and row one further above than upper column (up and left)
  - S2: Same row and column one further right than the row furthest right (right and down)
- Characteristics
  - Each granting set intersects with each granting set of the same scheme
  - Each granting set of the one scheme intersects with each granting set of the other scheme
  - All processes occur altogether in both schemes equally often in a granting set
- Thus, an alternating (or also random) usage of both schemes is possible → load balancing



Minimal Arrangement

Let K be the size of the granting set, then a minimal arrangement exists if there is a prime number p and a natural number m with

$$K\text{-}1 = p^m$$

The arrangement than has $n = K\,(K - 1) + 1$ processes

- K-1 = 1 = 1^1 $n = 3$ (here, we assume 1 as prime)
- K-1 = 2 = 2^1 $n = 7$
- K-1 = 3 = 3^1 $n = 13$
- K-1 = 4 = 2^2 $n = 21$
- K-1 = 5 = 5^1 $n = 31$
- K-1 = 7 = 7^1 $n = 57$
- …

For the size of the granting set holds:

$$K = \tfrac{1}{2}\,(1 + \sqrt{(4n - 3)}) = \lceil \sqrt{n} \rceil$$

Minimal Arrangement

$K = 2$

- $B_1 = \{1, 2\}$
- $B_3 = \{1, 3\}$
- $B_2 = \{2, 3\}$

$K = 3$

- $B_1 = \{1, 2, 3\}$
- $B_4 = \{1, 4, 5\}$
- $B_6 = \{1, 6, 7\}$
- $B_2 = \{2, 4, 6\}$
- $B_5 = \{2, 5, 7\}$
- $B_7 = \{3, 4, 7\}$
- $B_3 = \{3, 5, 6\}$

$K = 4$

- $B_1 = \{1, 2, 3, 4\}$
- $B_5 = \{1, 5, 6, 7\}$
- $B_8 = \{1, 8, 9, 10\}$
- $B_{11} = \{1, 11, 12, 13\}$
- $B_2 = \{2, 5, 8, 11\}$
- $B_6 = \{2, 6, 9, 12\}$
- $B_7 = \{2, 7, 10, 13\}$
- $B_{10} = \{3, 5, 10, 12\}$
- $B_3 = \{3, 6, 8, 13\}$
- $B_9 = \{3, 7, 9, 11\}$
- $B_{13} = \{4, 5, 9, 13\}$
- $B_4 = \{4, 6, 10, 11\}$
- $B_{12} = \{4, 7, 8, 12\}$

Token Based Algorithms

Simple Token Ring-Solution (Le Lann, 1977)

- Processes are arranged in a (logical) ring
- Access is controlled by circulating token
- Applicant waits for access until token reaches it
- Accessing process relays the token with the release
- Process without access intention relays the token directly

- Possible to use separate tokens for coordinating access to individual resources

Simple Token Ring-Solution

Advantages

- Simple, correct, fair algorithm
- No deadlocks
- No starvation
- Priorities are possible

Disadvantages

- Token is always on the way, under certain circumstances uselessly
- Thus, the message number per request is not limited
- Long waiting time with large number of processes

Token-Based Solution (Suzuki and Kasami, 1985)

- A requesting process sends a request with its sequence number to *all* other processes
  - This happens in a ring through a complete ring circuit
  - In another topology (complete meshing, tree etc.) through broadcast
- Each process P_i stores the highest currently received sequence number in a list R_i
- The token stores in a
  - Queue Q the processes waiting for the token
  - List L for each process the sequence number of the latest fulfilled request
- A process P_i can determine which requests have not yet been served by comparing of R_i with L when receiving the token

Token-Based Solution

If a process P_i receives the token, it does the following:

- Accesses if it wants to
- Sets $L[i] := R_i[i]$ (enters its current sequence number as its last access)
- Attaches each process P_j (order in increasing sequence numbers) not part of Q to the end of Q for which applies
  $R_i[j] > L[j]$ (local stored sequence number for process j is larger than seq.num. in list of token => request has not been served yet)
- Deletes itself from Q
- If Q is not empty afterwards, the process sends the token
  - to the next process (ring),
  - to the first process in Q (complete meshing) or
  - to the next process in direction of the first process in Q (different topology)
- Otherwise it only sends the token on, if it receives a request from a process P_j whose sequence number is larger than $L[j]$

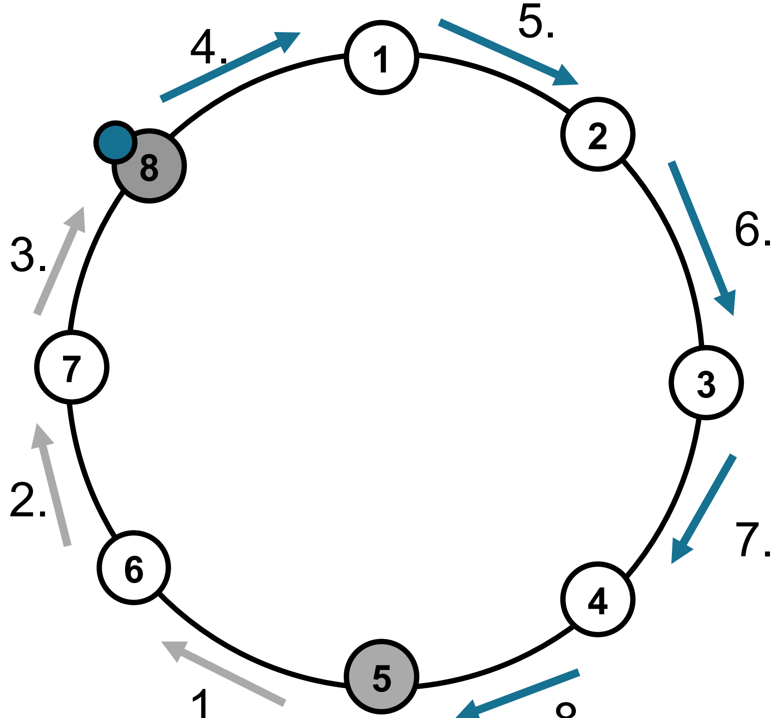
Solution with a Ring

Most current request

Q	L	R_8
5	1, 0	1, 0
	2, 0	2, 0
	3, 1	3, 1
	4, 0	4, 0
	5, 3	**5, 4**
	6, 0	6, 0
	7, 0	7, 0
	8, 5	8, 5

Processes waiting for access

Last fulfilled request.



REQUEST(5, 4)

1. A request does not need to be relayed if it meets the *resting* token.

2. The algorithm can be simplified to a great extent if there are no overtakes.

3. Maximal 2*n*-1 messages per access are needed in the physical topology

All depicted states after 3.

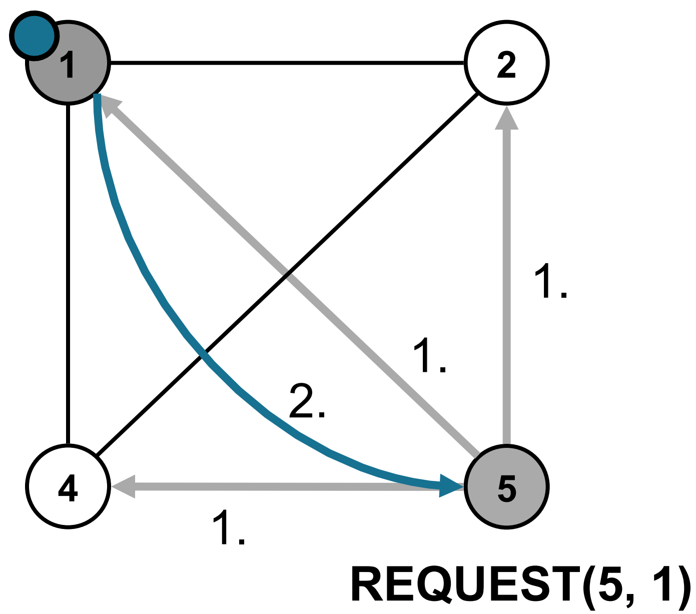
Solution with Complete Meshing

Exactly *0* or *n* messages are needed in the physical topology.

Q
5

L
1, 1
2, 0
3, 0
4, 0
5, 0
6, 0
7, 0
8, 0

R_1
1, 1
2, 0
3, 0
4, 0
5, 1
6, 0
7, 0
8, 0



All depicted states after 1.

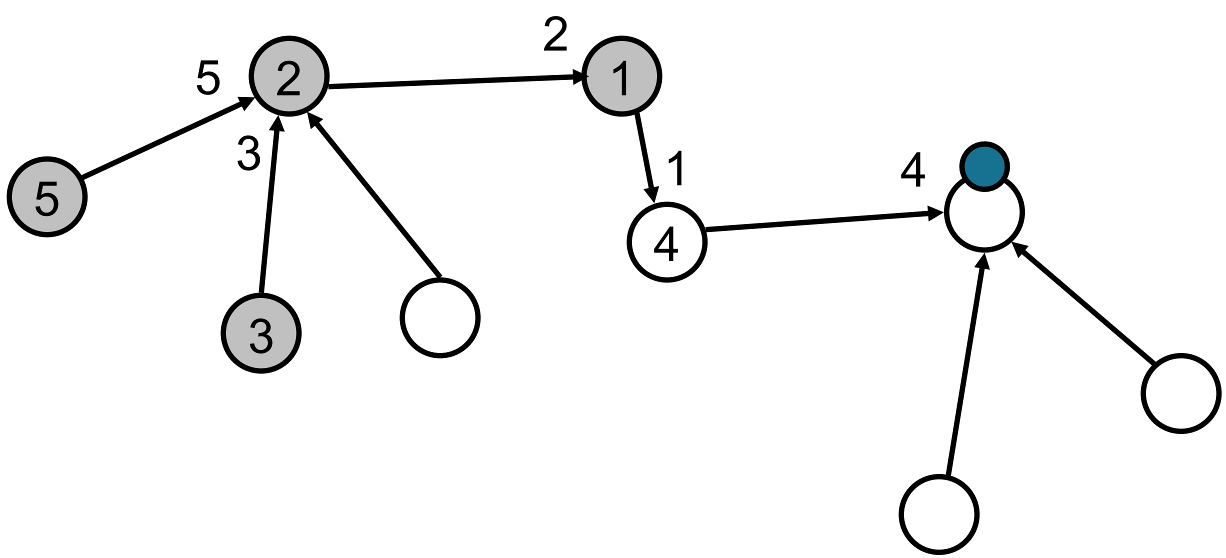
Lift Algorithm (Raymond, 1989)

- Uses a spanning tree for the *selective* relay of the request in direction to the token (instead of sending the request to all processes)
- The edges of the spanning tree have a state; each can point in one of two directions
- The token wanders against the arrow direction and thereby turns around the direction of each passed edge
- A process that wants the token sends the request over its *outgoing* edge
- If a process has received a request, it sends a request in the direction of the token (once)

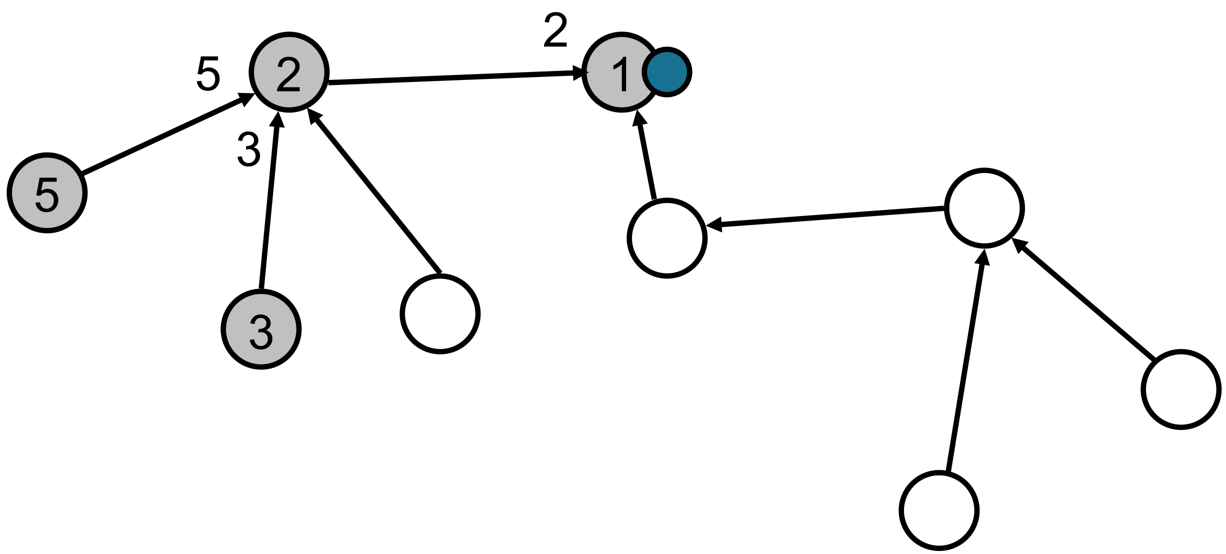
Lift Algorithm

- Each process remembers the processes from which it has received a request
- If a process receives the token
  - It relays it in one of the requesting directions
  - If there are more requests from other directions, it sends a request after the token
- To ensure fairness, a process must not ignore a requesting direction arbitrarily often

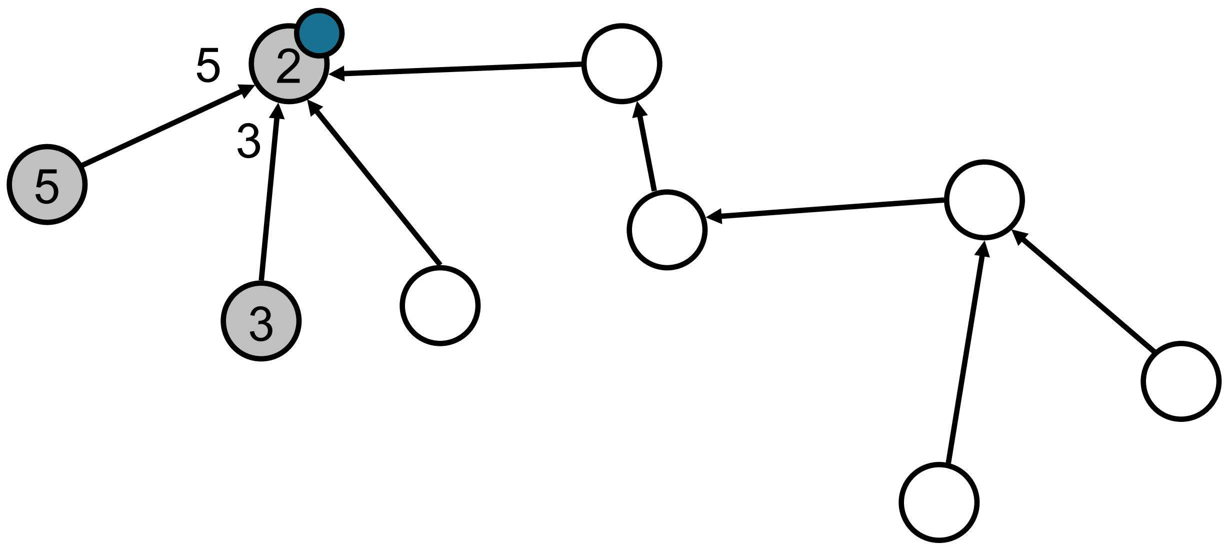
Lift Algorithm



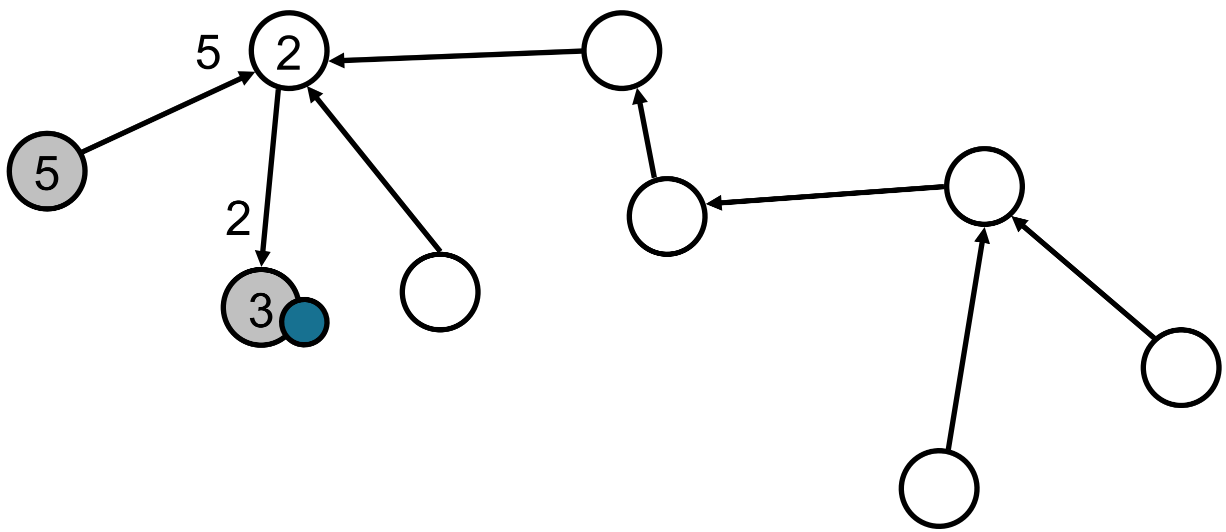
Lift Algorithm



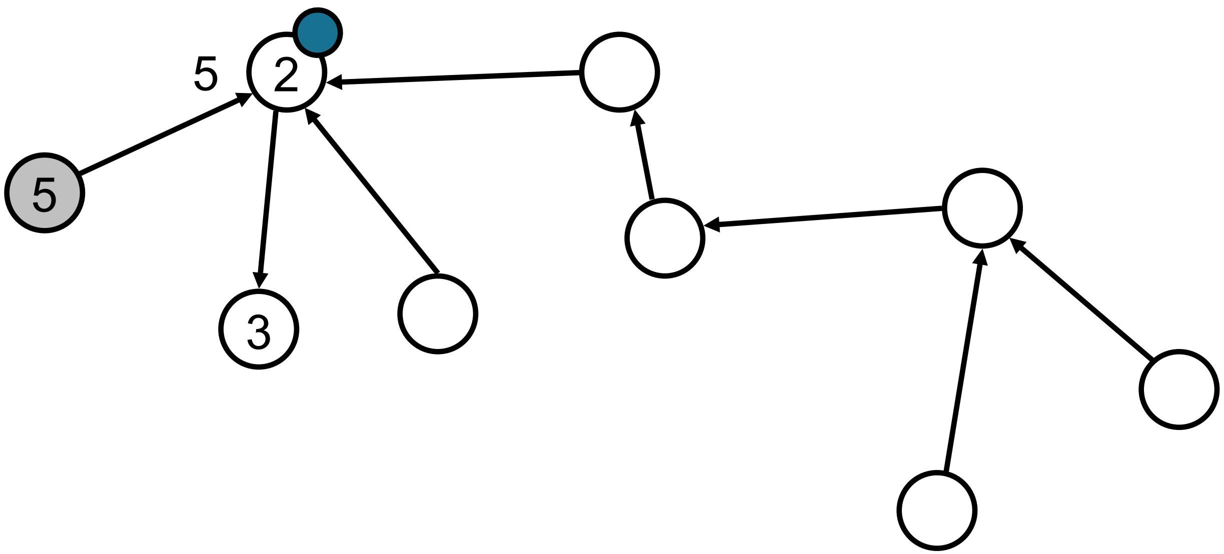
Lift Algorithm



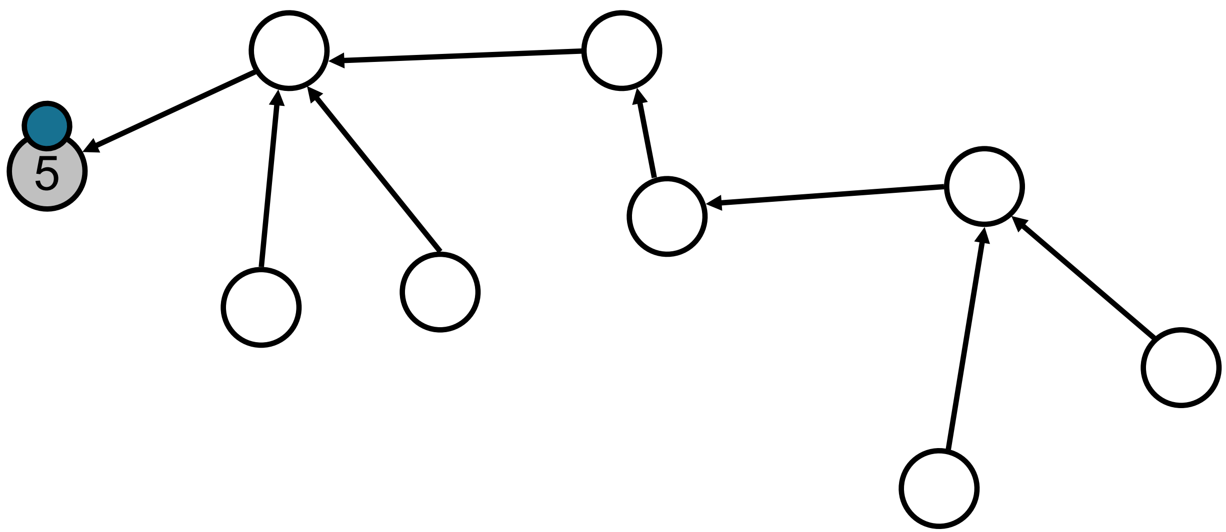
Lift Algorithm



Lift Algorithm



Lift Algorithm



Lift Algorithm

- Invariant: From each process a directed path leads to the token
- In a *k*-ary balanced tree the maximal path length between arbitrary processes is $O(log_k\ n)$
- Accordingly, only $O(log_k\ n)$ messages per access are needed
- Start state: Winner of an election gets the token and creates a spanning tree with edges directed towards itself
  - Both can be achieved simultaneously by using the echo algorithm
- Procedure can be generalized for arbitrarily connected topologies

Comparison of the Algorithms

Comparison of Message Complexity per Access

Procedure	Message Complexity on Logical Topology
Token Ring	1 ... ∞
Simple Broadcast	$3\ (n - 1)$
Improved Broadcast	$2\ (n - 1)$
Improved Token Ring	$0 \ldots 2n - 1$
Mesh Arrangement	$O(\sqrt{n})$
Lift Algorithm on *k*-ary Tree	$O(log_k\ n)$
Central Manager	3

Literature

1. L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed Environment. Communications of the ACM, 21:558--564, July 1978.
2. G. Ricart and A. K. Agrawala. An Optimal Algorithm for Mutual Exclusion in Computer Networks. Communications of the ACM, 24(1):9--17, 1981.
3. M. Maekawa. A $\sqrt{N}$ Algorithm for Mutual Exclusion in Decentralized Systems. ACM Transactions on Computer Systems, 3(2):145--159, 1985.
4. K. Raymond. A Tree-Based Algorithm for Distributed Mutual Exclusion. ACM Transactions on Computer Systems, 7(1):61--77, 1989.
5. W. S. Luk and T. T. Wong. Two New Quorum Based Algorithms for Distributed Mutual Exclusion. In Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS '97), pages 100--107, 1997. IEEE Computer Society.
6. I. Suzuki and T. Kasami. A distributed mutual exclusion algorithm. ACM Transactions on Computer Systems, 3(4):344--349, 1985.
7. A. S. Tanenbaum and M. van Steen. Distributed Systems: Principles and Paradigms. Prentice Hall, 2002. Chapter 5, pages 262--270
8. N. Lynch. Distributed Algorithms. Morgan Kaufmann, 1996. Chapter 10
9. G. Coulouris, J. Dollimore, and T. Kindberg. Distributed Systems: Concepts and Design. Addison-Wesley, 3rd edition, 2001. Chapter 11.2, pages 423--431