Telecooperation Lab
Prof. Dr. Max Mühlhäuser

TK1: Distributed Systems - Programming & Algorithms

Chapter 3: Distributed Algorithms

Section 3: Coordination – Failure Detection, Mutex, Election

Lecturer: **Prof. Dr. Max Mühlhäuser**

Coordination

Coordination Problems in Distributed Systems

Idea: instead of solving time & state problems in general, we may devise algorithms for a specific class of problems – still quite general

1. failure detection: know in asynch. net whether peer is dead or alive
2. mutual exclusion: never >1 process granted access to a shared resource in a critical section at the same time
3. election: for master-slave systems: system elects a master (either at boot up time or when the master fails)
4. multicast: sending to a group of recipients
   - reliability of multicast (correct delivery, only once, etc.)
   - order preservation
5. consensus in the presence of faults (arbitrary faults → 'byzantine' problems):
   - know whether ACK was received over an unreliable communication medium
   - know whether peer process knows about one's own intentions in the presence of a non-confidential communication channel

Items 4+5 treated in separate section 'cooperation'

Failure Detection

- Failure detection means the capability to decide whether a particular process has crashed or not
- Unreliable failure detection
  - Distinguishes suspected processes from unsuspected processes
    - Unsuspected: failure is unlikely (e.g., recently received communication from unsuspected process)
      - May be inaccurate
    - Suspected: indication that process failed (e.g., no message received for some time)
      - May also be inaccurate (process has not failed, but link is down or process is much slower than expected)
- Reliable failure detection
  - Unsuspected: potentially inaccurate as above
  - Failed process (accurate determination)

Failure Detection

- Unreliable failure detection in asynchronous systems
  - Suspected
  - Unsuspected
- Reliable failure detection *only* in synchronous systems
  - Failed
  - Unsuspected
- Implementation
  - Every T seconds P sends to all: "P is here"
  - Bound on message transmission time:
    - Asynchronous system: Estimate E
    - Synchronous system: Absolute bound A
  - Problem: how to calibrate E?
    - E too small → intermittent net performance downgrades will lead to suspected nodes, or
    - E too large → crashes remain unobserved (crashed nodes will be fixed before timeout expires)
  - Solution: Adjust E based on observed net latencies

no "P alive" msg in T + E sec

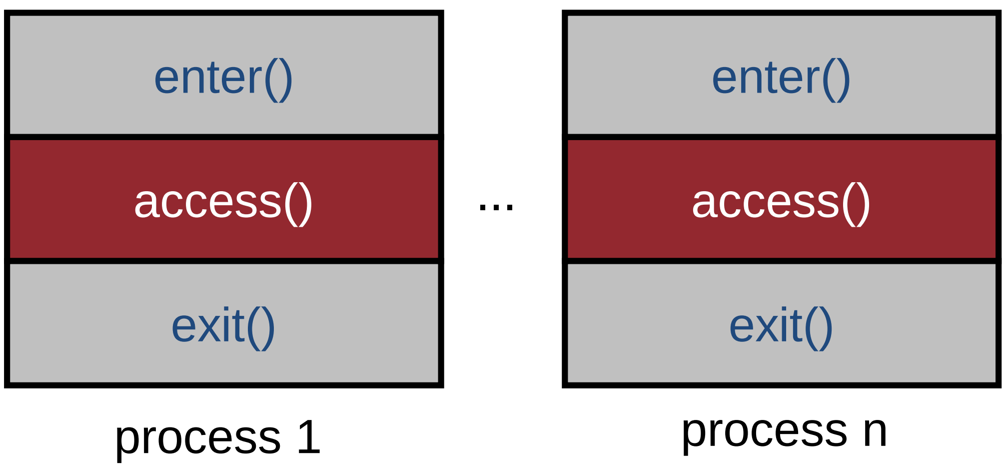
no "P alive" msg in T + A sec

Mutual Exclusion (Mutex)

- Problem: How to give a single process temporarily a privilege?
  - Privilege = the right to access a (shared) resource
  - Resource = file, device, window,…
- Assumptions
  - Clients execute the mutual exclusion algorithm
  - The resource itself might be managed by a server
  - Reliable communication
- Basic requirements:
  - ME1 (Safety): At most one process may execute in the shared resource at any time
  - ME2 (Liveness): A process requesting access to the shared resource is eventually granted it
  - ME3 (Ordering): Access to shared resource should be granted in happened-before order (*if* happened-before than granted before) – desired, not mandatory
    - Note: ordering is *one* possible fairness property, like “no starvation”
    - Note: “... in order-of-time…” would be impossible to implement!

Mutual Exclusion

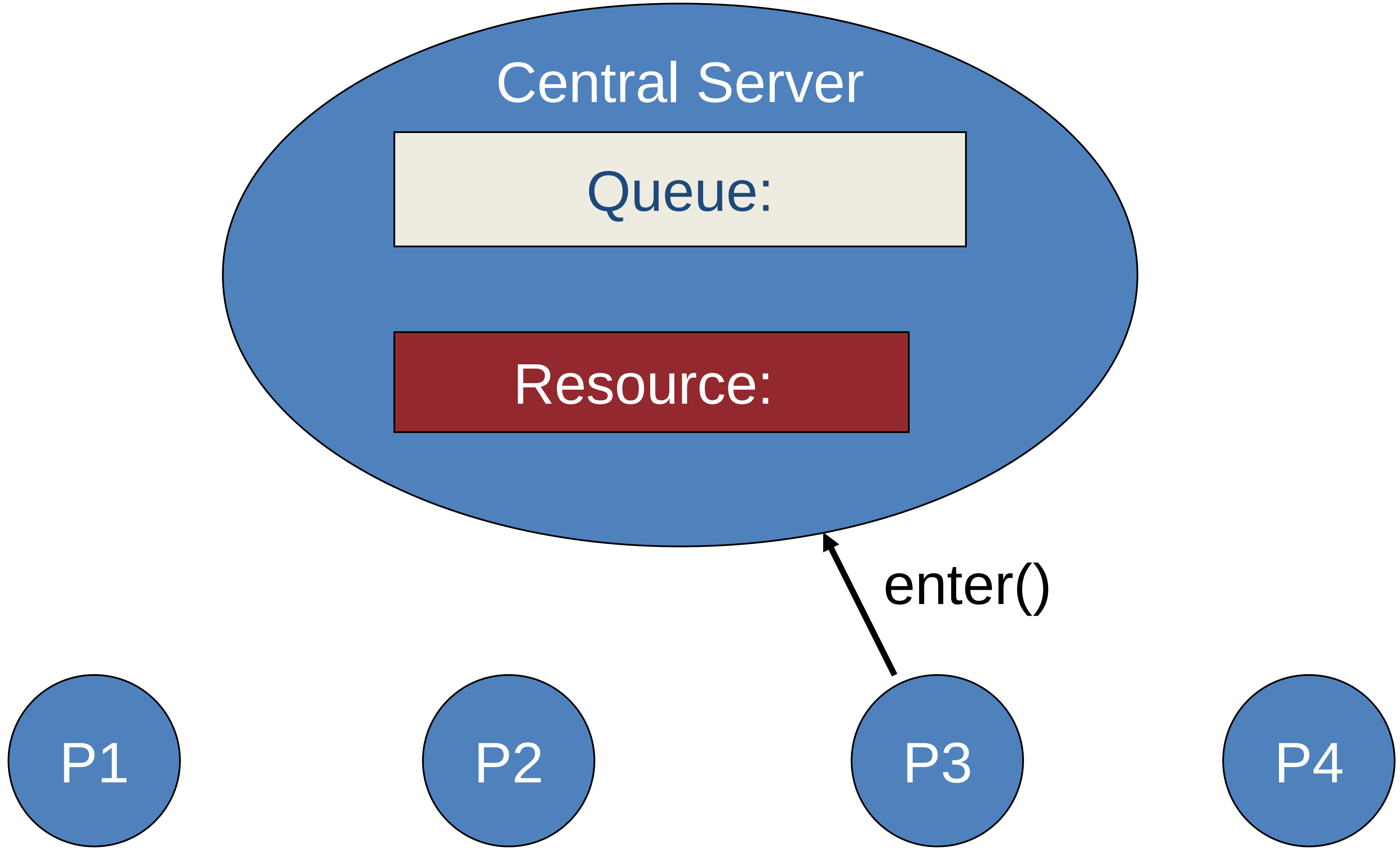
- Solutions:
  - Central server algorithm
  - Distributed algorithm using logical clocks
  - Ring-based algorithm
- Evaluation
  - Bandwidth (= #messages to enter and exit)
  - Client delay (incurred by a process at enter and exit)
  - Synchronization delay (delay between exit and enter)



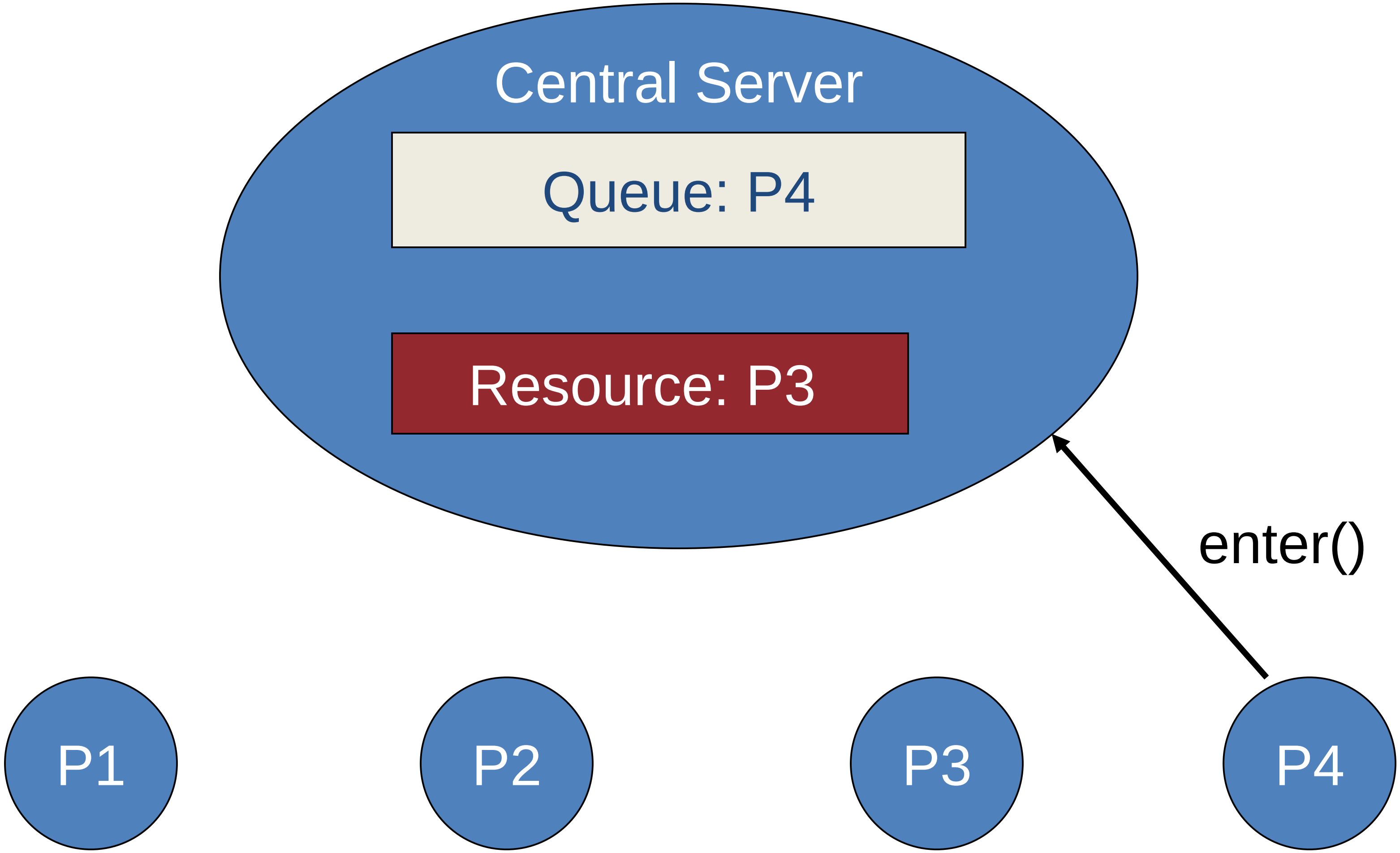
Mutex: Central Server Algorithm

- Central server offering 2 operations:
  - enter(): if resource free then operation returns without delay
    else request is queued, return of enter() is delayed
  - exit(): if request queue is empty then resource is marked free
    else enter for a selected request is executed
- Evaluation:
  - ME3 satisfied!
  - Performance:
    - Single server is performance bottleneck
    - Enter critical section: 2 messages (enter() + acknowledgement)
    - Synchronization: 2 messages between the exit of one process and the enter of next process
  - Failure:
    - Central server is single point of failure
    - What if a client, holding the resource, fails?
    - Reliable communication required

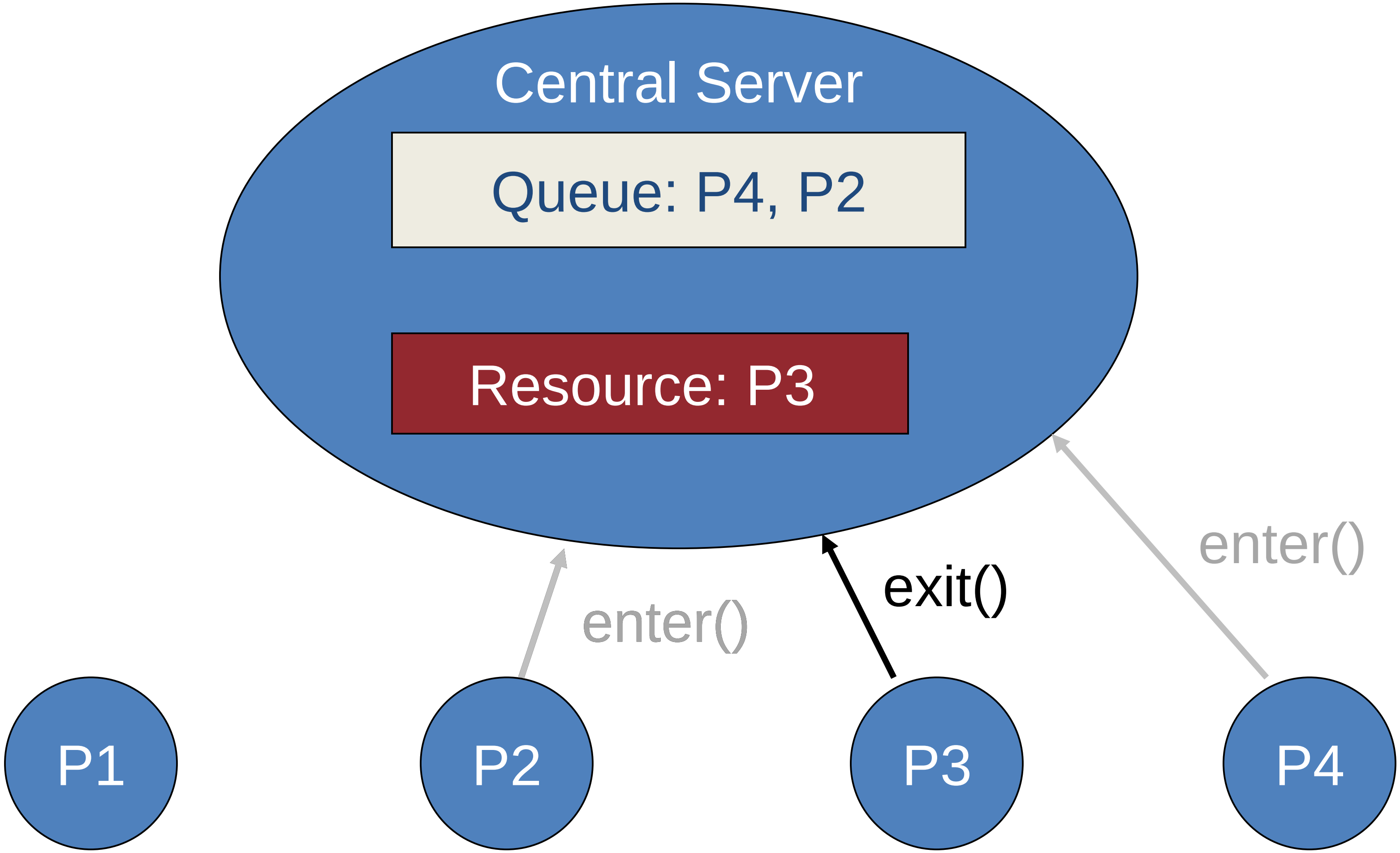
Mutex: Central Server Example



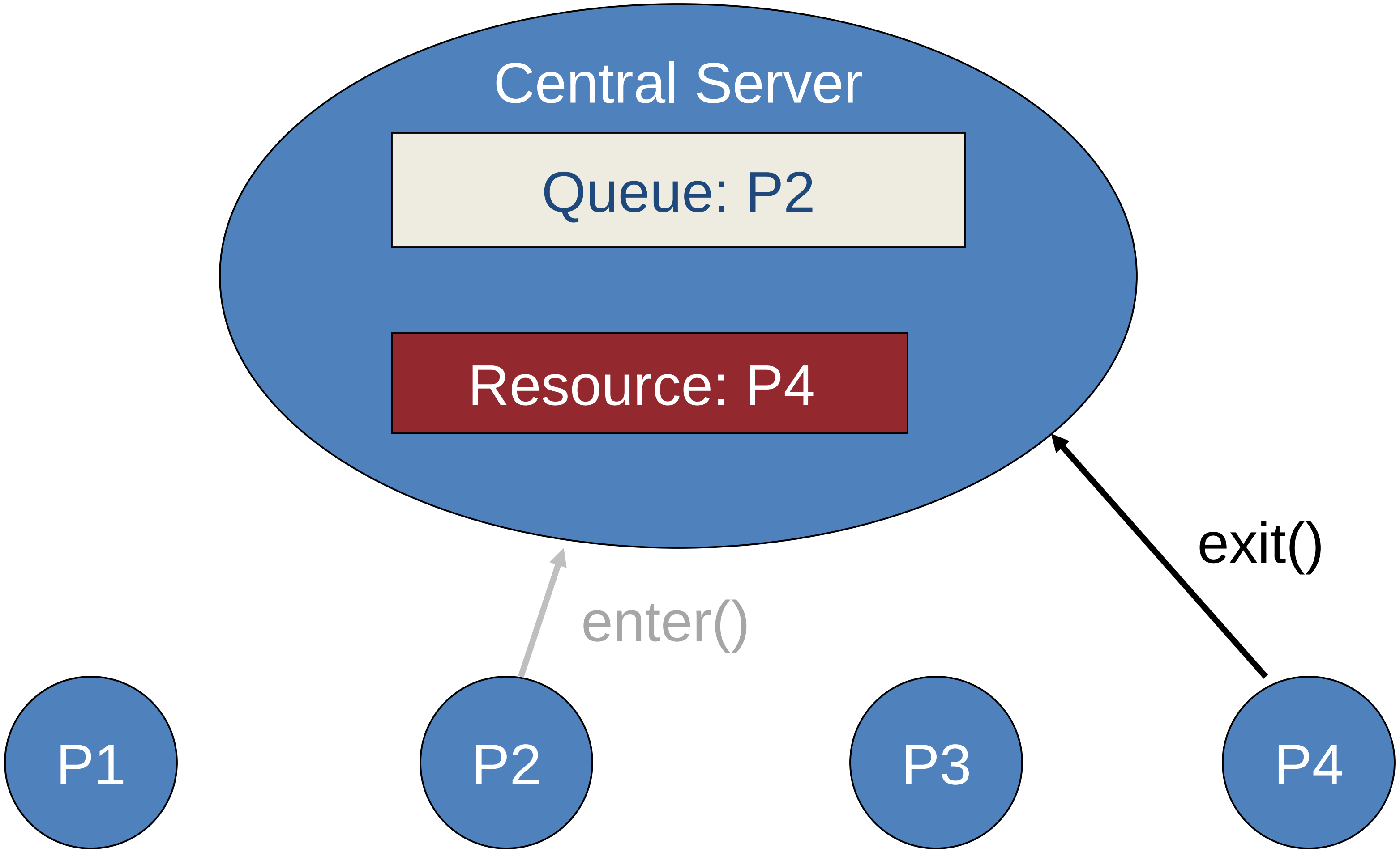
Mutex: Central Server Example



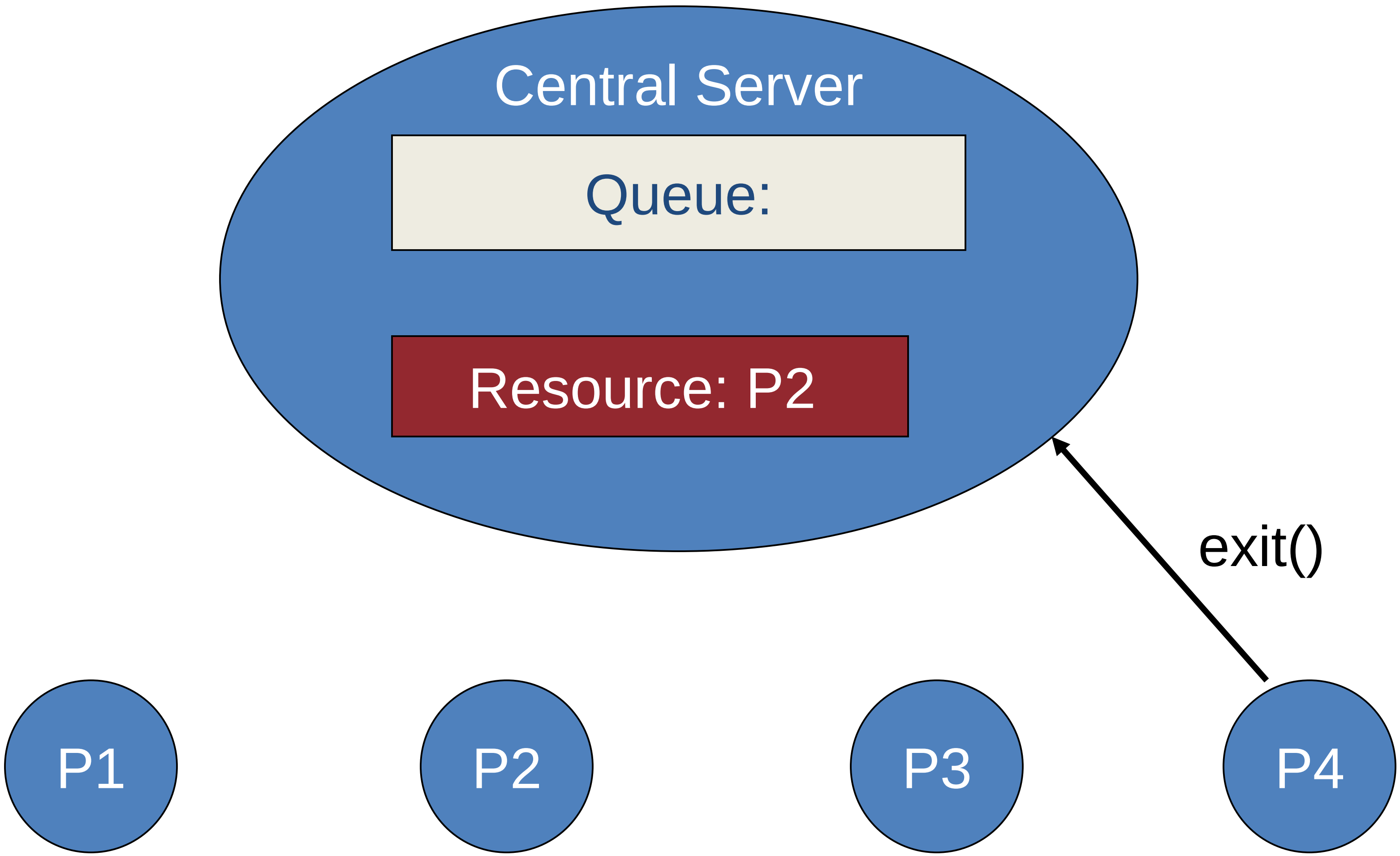
Mutex: Central Server Example



Mutex: Central Server Example



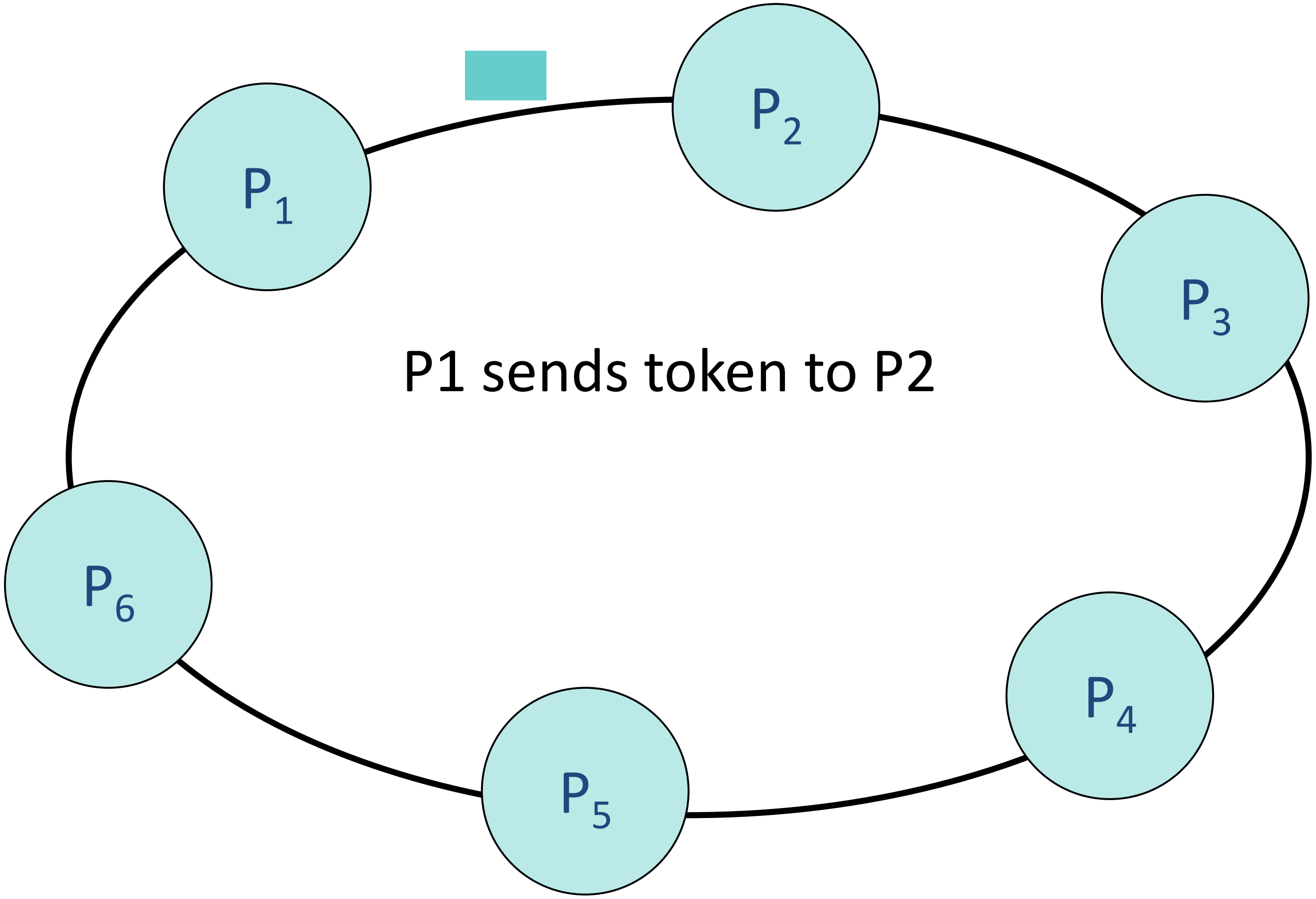
Mutex: Central Server Example



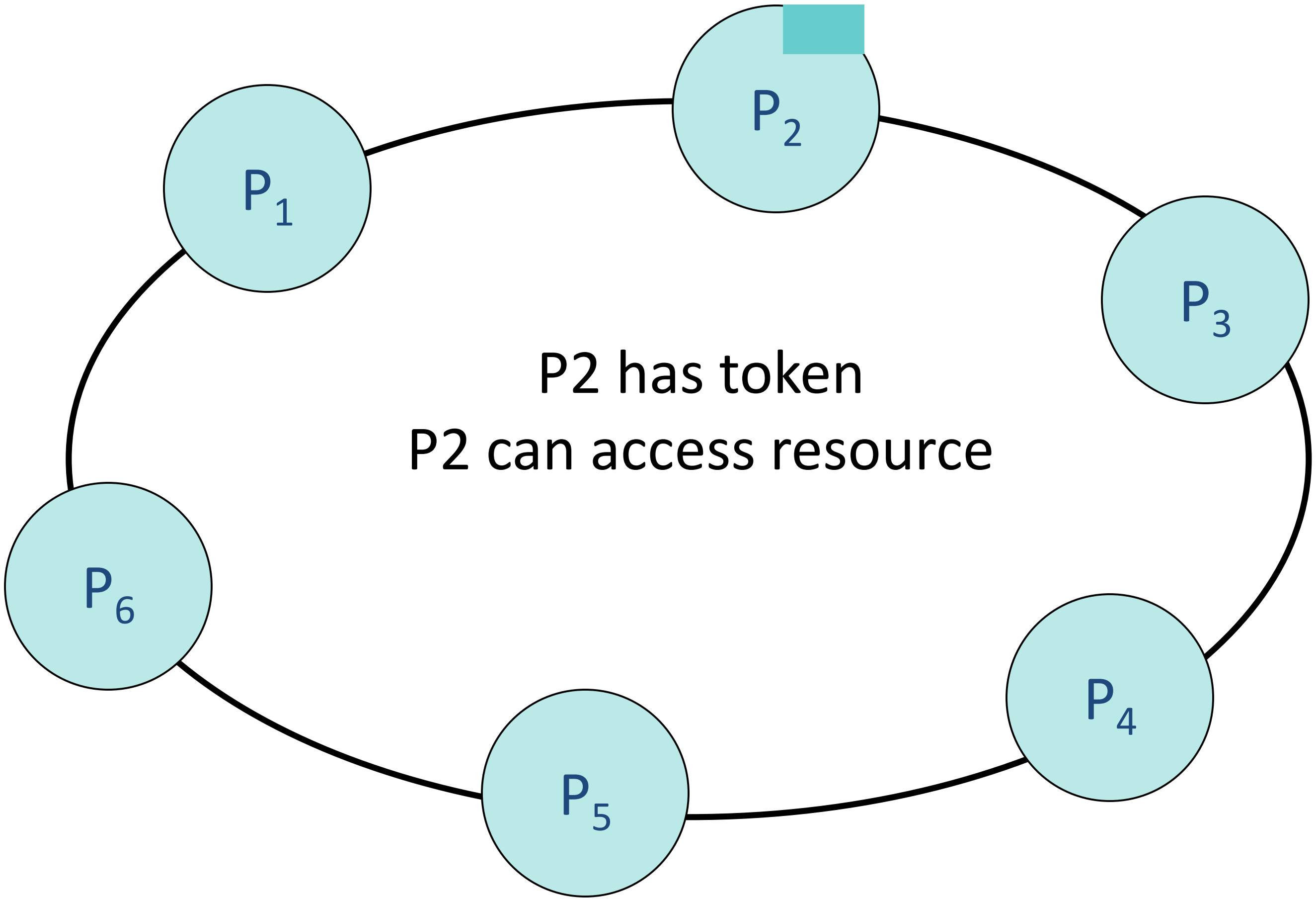
Mutex: Ring-Based Algorithm

- All processes arranged in a unidirectional ring
- Logical, not necessarily physical link
  - Every process p_i has connection to process p_{i+1} (mod N)
- Token passed in ring
- Process with token has access to resource
- Evaluation:
  - ME3 not satisfied
  - Efficiency
    - High when high usage of resource
    - High overhead when very low usage
  - Failure
    - Process failure: Loss of ring!
    - Reliable communication required

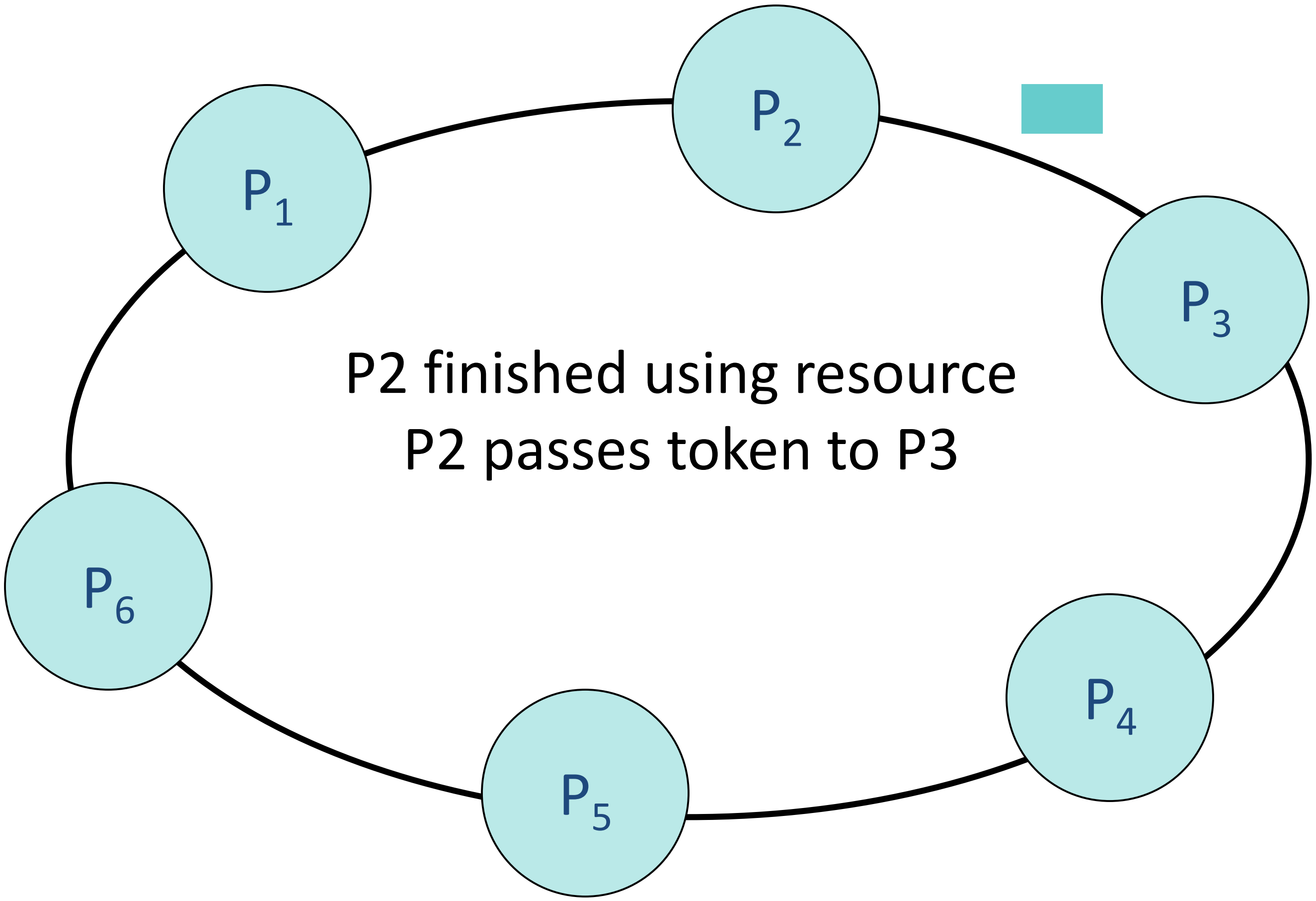
Mutex: Ring-Based Example



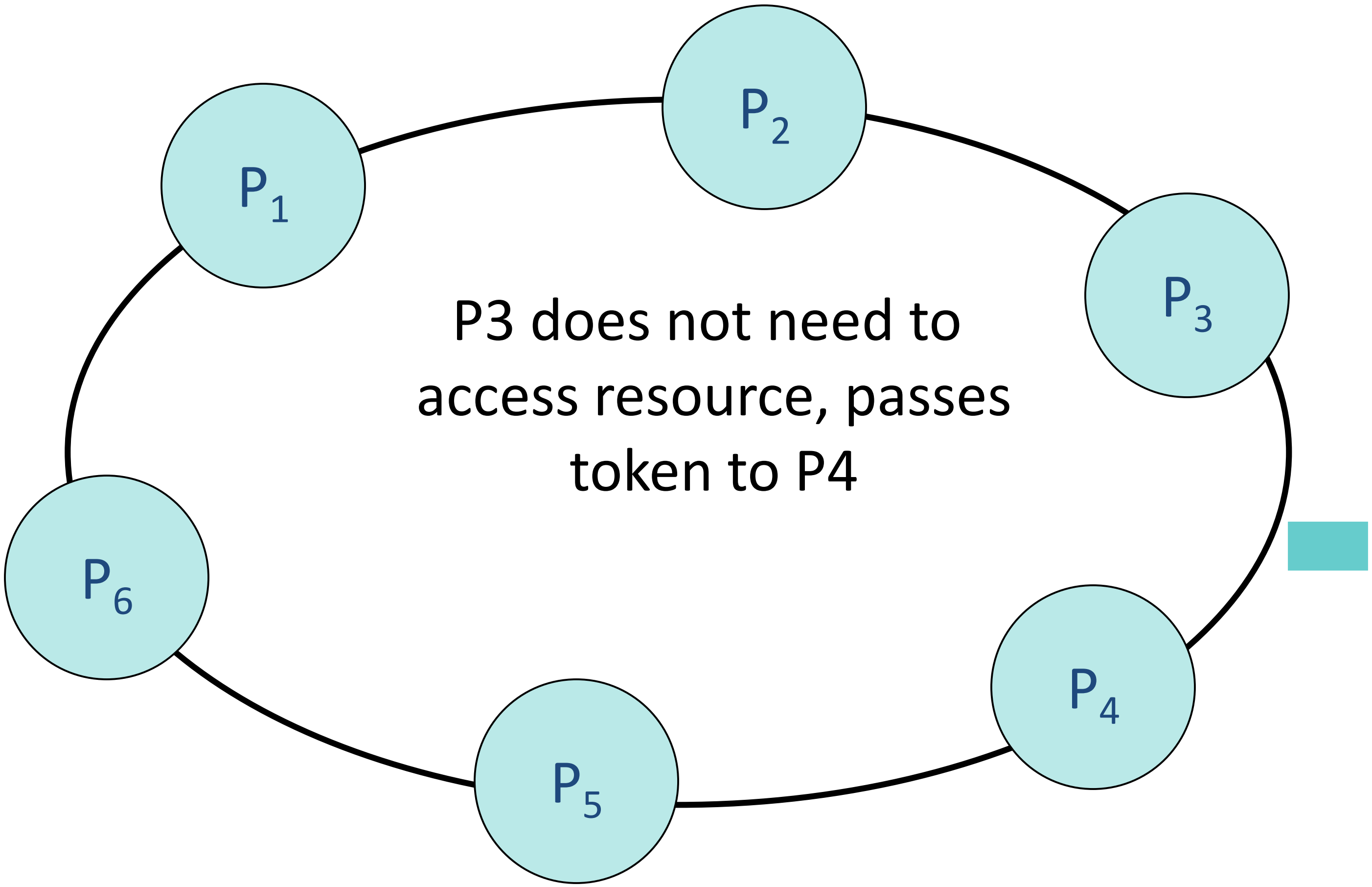
Mutex: Ring-Based Example



Mutex: Ring-Based Example



Mutex: Ring-Based Example

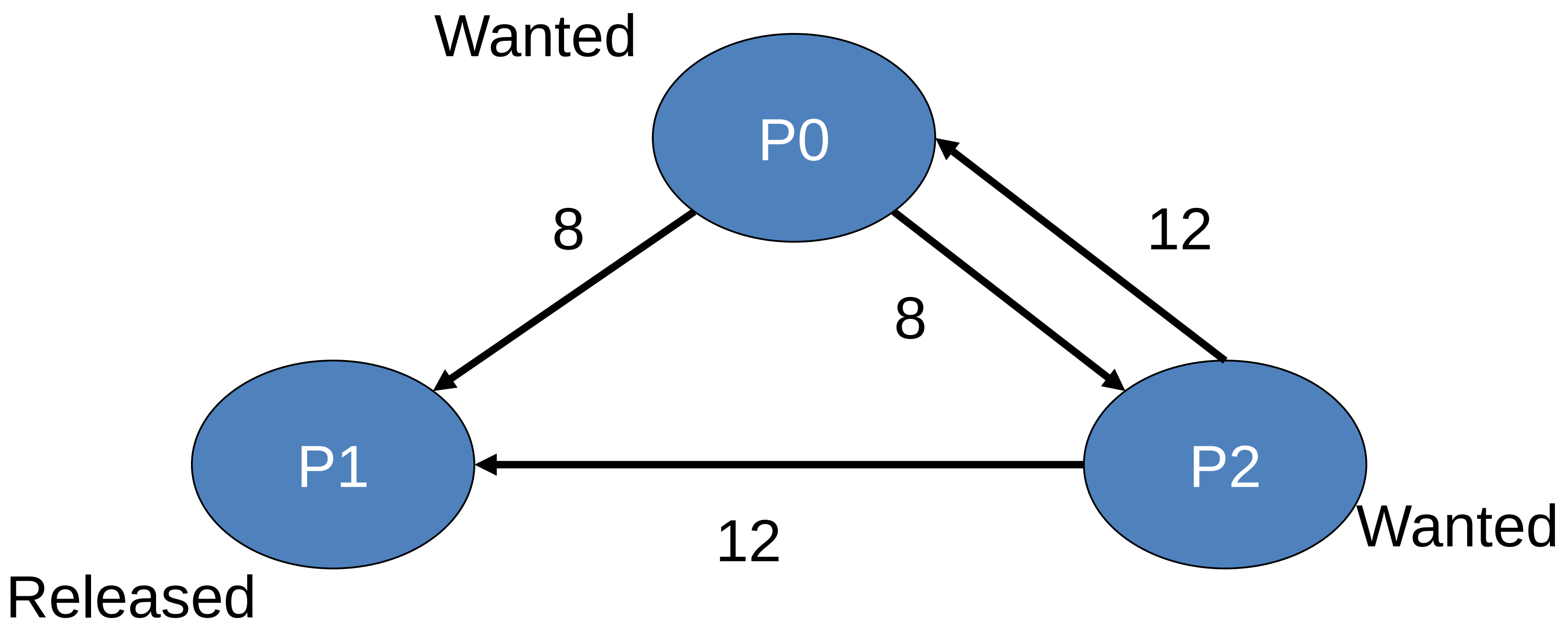


P3 does not need to access resource, passes token to P4

Mutex via Logical Clocks

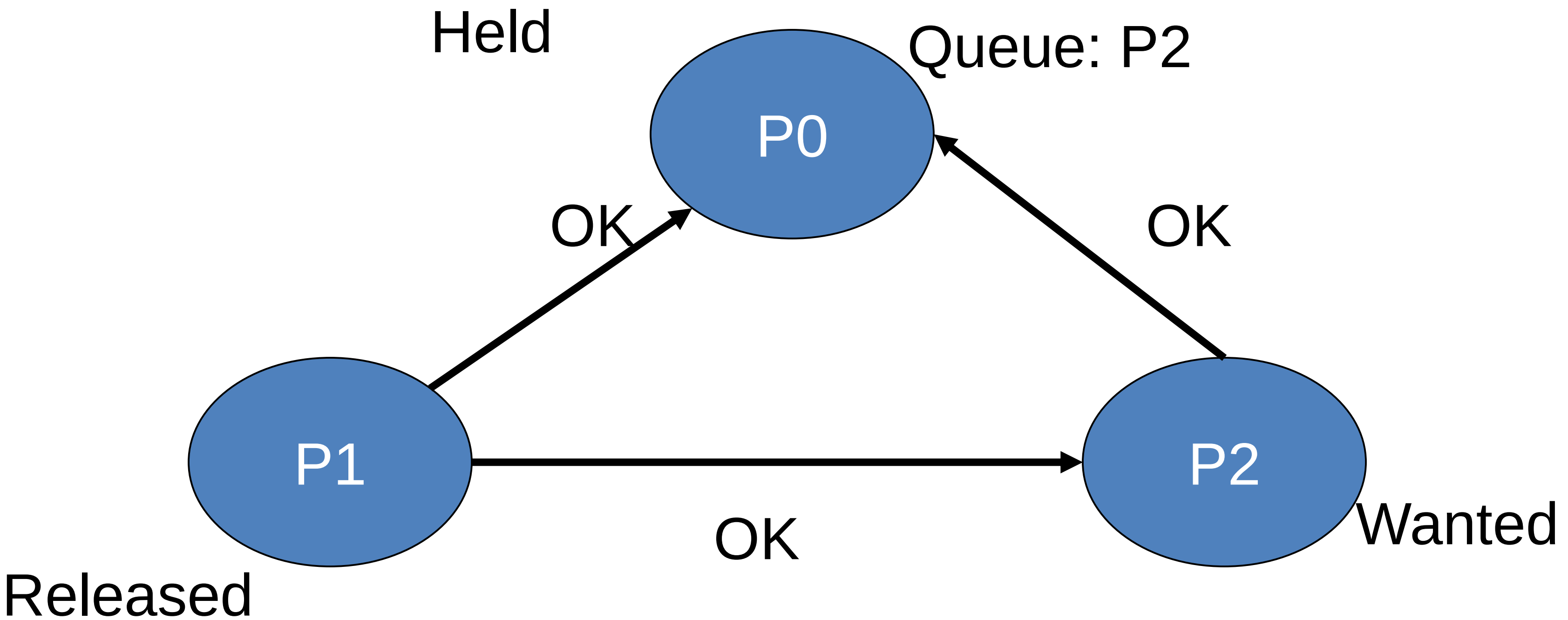
- Distributed agreement algorithm
  - Broadcast requests to all participating processes
  - Use resource when all other participants agree (= reply received)
- Processes
  - Keep logical clock; included in all request messages
  - Behave as finite state machine
  - States: Released / Wanted / Held
- Sketch of Algorithm:
  - If request is broadcast and state of all other processes is RELEASED, then all processes will reply immediately → requester will obtain entry
  - If at least one process is in state HELD, that process will not reply until it has left critical section, hence mutual exclusion
  - If more than 2 processes request at the same time, process with lower timestamp will be the first to get N-1 replies
  - In case of equal timestamps, process with lower ID wins

Mutex: Logical Clocks Example



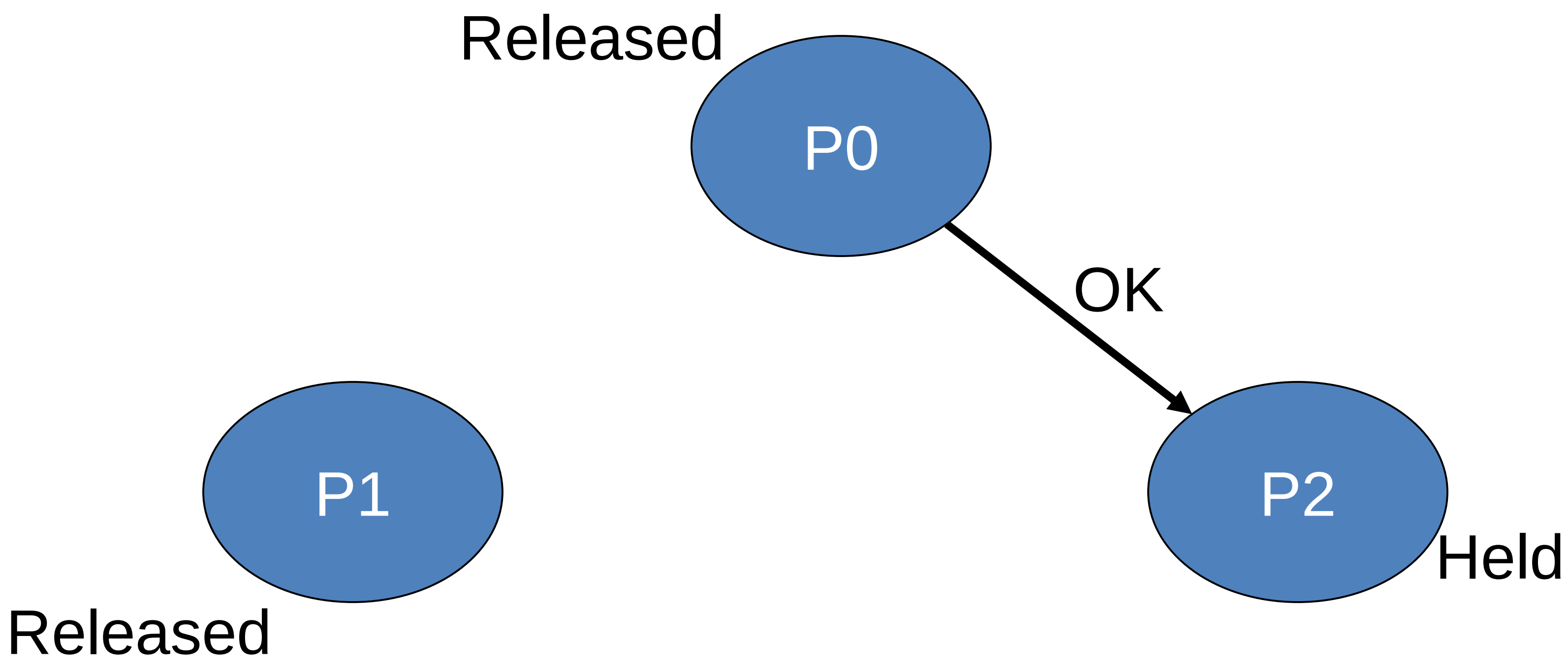
- P0 and P2 want to access resource
- P0 and P2 send request to all others
- Request contains their local timestamp

Mutex: Logical Clocks Example



- P1 has no interest in resource, sends OK to all
- P2 sees it has bigger timestamp, sends OK to P0
- P0 receives OK from all, gets resource
  - Queues request from P2

Mutex: Logical Clocks Example



- When P0 is finished, it sends OK to P2
- P2 can now access resource

Mutex: Logical Clocks Evaluation

- Performance:
  - Expensive: 2 * (N - 1) messages to get resource
  - Synchronization delay: 1 message to pass resource to another process
- Protocol improvements
  - Repeated entry of same process without executing protocol
  - Get OK from a majority, not all processes
- Problems
  - Each process must know all other processes
  - Crash of any process
  - Reliable communication required

Mutex: Intermediate Summary

Algorithm	#of msgs per entry/exit → "required bandwidth"	Delay before entry (msgs) → "performance"	Problems
Central server	3	2	Central server crash
Ring-based	1 to ∞	0 to n - 1	Lost token, process crash
Logical clock (Ricart&Agrawala)	2 * (n - 1)	1	Process crash

- Central server is simplest and most efficient
  - Solution: Let the server managing the resource perform concurrency control and mutex
  - Gives more transparency for the clients
- Ironic remark: Distributed algo's more susceptible to crashes than centralized
- BELOW: use *voting algorithm* for mutex instead → desired properties:
  - ME1 (Safety): ≤ 1 winner if several procs want to be "voted" to get access
  - ME2 (Liveness): a proc requesting access to shared resource is eventually granted it
  - ME3 (ordering): access to shared resource should be granted in happened-before order (*if* happend-before than granted before) [not fulfilled by all algo's]

Mutex via Voting Algorithm

Maekawa's Voting Algorithm:

- Observation from 'real world' elections:
  - in order to have ≤ 1 winner, not all processes have to agree
  - absolute majority is sufficient
  - even relative majority is sufficient IF:
    - set of proc's split up into overlapping subsets ("voting sets")
    - a proc receives all votes from "its" voting set
  - Attention: 'election algorithms' are fundamentally different, see later
- model
  - processes $p_1, .., p_N$
  - voting sets $V_1, .., V_N$ chosen such that $\forall$ i,k and for some integer M:
    $p_i \in V_i$ (optimization: "vote for myself" does not require network communication)
    $V_i \cap V_k \neq \emptyset$ (some overlap in every voting set)
    plus, ideally, fairness 1 (equal effort) : $|V_i| = K$ (all voting sets have equal size)
    plus, ideally, fairness 2 (equal responsibility) : each p_k member of same # of voting sets M

- obviously: in ideal case, if N proc.s are members in M sets each, then:
  N*M memberships, i.e. size of each (of N) sets is (N*M)/N = M, so: K = M
- quest for optimal solution (set sizes etc.), see below

Mutex: Voting Algorithm (cont.)

TECHNISCHE UNIVERSITÄT DARMSTADT

Algorithm Sketch:

- to obtain entry to critical section, p_i sends request messages to all K-1 other members of voting set V_i
- cannot enter until K-1 replies received
- when leaving critical section, send release to all members of V_i
- when receiving request
  - if state = HELD or already replied (voted) since last request
    - then queue request
  - else immediately send reply
- when receiving release
  - remove request at head of queue and send reply

Mutex: Voting Algorithm (cont.)

```
On initialization
    state := RELEASED; voted := FALSE;
For p_i to enter the critical section
    state := WANTED;
    Multicast request to all processes in V_i – {p_i};
    Wait until (number of replies received = (K – 1));
    state := HELD;
On receipt of a request from p_i at p_j (i ≠ j)
    if (state = HELD or voted = TRUE)
    then
        queue request from p_i without replying;
    else
        send reply to p_i;
        voted := TRUE;
    end if
For p_i to exit the critical section
    state := RELEASED;
    Multicast release to all processes in V_i – {p_i};
On receipt of a release from p_i at p_j (i ≠ j)
    if (queue of requests is non-empty)
    then
        remove head of queue – msg was received from p_k, say;
        send reply to p_k;
        voted := TRUE;
    else
        voted := FALSE;
    end if
```

pseudo code is a bit flaky wrt. own vs. foreign requests: vote for “myself” and for foreign node are mutually exclusive

Mutex: Voting Algorithm (cont.)

Optimization: Goal „minimize K while assuring mutual exclusion“

- optimal solutions may exist if N is of the form $N= K * (K-1) +1$ (N= 3, 7, 13)
- for N=7/K=3, e.g.: (1,2,3), (2,4,5), (3,4,6), (4,1,7), (5,1,6), (6,2,7), (7,3,5)
- mathematically proven to exist if $k=K-1=p^i$ (p prime, i integer; e.g., N=7, K=3, k=2)
  - “finite projection plane FPP for N nodes, of degree k … deep math!”
- mathematically proven NOT to exist if k-1 or k-2 devisable by 4 and $k \neq a^2+b^2$
- else: problem unsolved (exhaustive search. N=111, k=10: negative! took 1000h Cray1-CPU)
- if solution for N is known, solution for N’ < N (*without* fairness) may be:
- take solution for N, then: for any p_i with i > N’: replace by existing node consistently

- due to problems with FPP, authors gave approximation
  - derive V_i so that $| V_i | = 2\sqrt{N} - 1$
    - note: satisfies fairness only if N is an integer square
    - example: N=3 → $|V_i| = 5$
    - place processes in a $\sqrt{N}$ by $\sqrt{N}$ matrix
    - let V_i the union of the row and column containing p_i

1	2	3
4	5	6
7	8	9

V_5 =(2,4,5,6,8)

  - Properties: satisfies ME1
    - if possible for two proc‘s to enter critical section, then proc‘s in non-empty intersection of their voting sets would have both granted access
    - impossible: all proc‘s make at most one vote after receiving request

Mutex: Voting Algorithm (cont.)

- Deadlocks: in above algo., deadlocks are possible
  - consider N=7 as above, nodes 1, 2, and 5 want access to resource „concurrently“
  - possible to construct cyclic wait graph
    - 1 gets OK from 3, but not from 2
    - 2 gets OK from 4, but not from 5
    - 5 gets OK from 6, but not from 1
- Deadlock Avoidance: possible by modification of algo.
  - use of logical clocks
  - processes queue requests in happened-before order
  - means that ME3 is also satisfied
- Performance
  - bandwidth utilization
    - $2\sqrt{N}$ per entry, $\sqrt{N}$ per exit, total $3\sqrt{N}$ is better than Ricart and Agrawala for N>4
  - client delay
    - same as for Ricart and Agrawala
  - synchronization delay
    - round-trip time instead of single-message transmission time in Ricart and Agrawala

Mutex: Voting Algorithm (cont.)

notes on fault tolerance:

- none of these algorithms tolerates message loss
- ring-algorithms cannot tolerate single crash failure
- Maekawa’s algorithm can tolerate some crash failure
  - if process is in a voting set not required, then rest of system not affected (may be exploited further)
- Central-Server: tolerates (only!) crash failure of node that has neither requested access nor is currently in the critical section
- Ricart & Agrawala algo. can be modified to tolerate crash failures by the assumption that a failed process grants all requests immediately
  - requires reliable failure detector

wide-spread (but not unique!) summary:

- algorithms are expensive and not practical
- algorithms are extremely complex in the presence of failures
- better solution in most cases:
  - let the server, managing the resource, perform concurrency ctrl.
  - gives more transparency for the clients

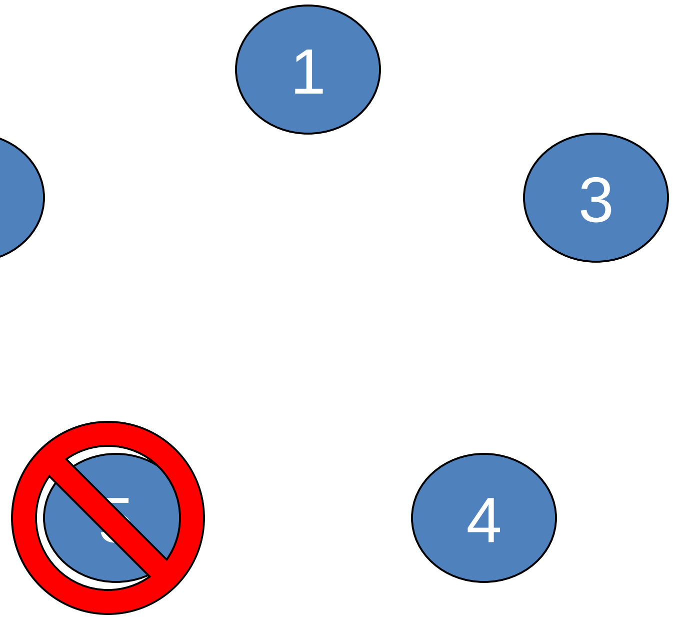
Election

- Many distributed algorithms require coordinator
  - As to Mutex with central server: select server (…replacement after failure)
- It does not matter which process is coordinator, as long as one of them is
- Assume that each process has a unique number
  - For example, network address + process number
- Idea of ‘election algorithm’:
  - Find currently running process **with highest number**, designate it as coordinator
  - *Attention: different from intuitive understanding, different from voting!*
- Further assumption: Each process knows the numbers of all other processes, but does not know which processes are currently running

Bully Algorithm

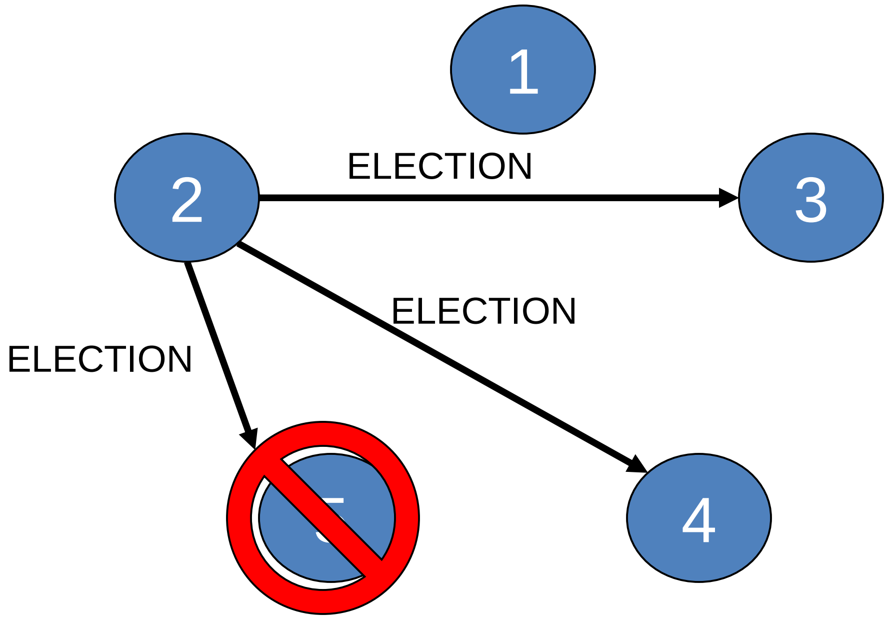
- If a process notices that coordinator is not responding, it starts an election
- Process P holds an election as follows:
  1. P sends ELECTION msg. to every process with a higher number (some of them may be down)
  2. If nobody answers ‘OK’, P wins election and becomes the new coordinator
  3. If any of the higher-ups answers, it takes over. P’s job is done
- When process receives ELECTION from lower-numbered process: replies ‘OK’
  - Indicates that it is alive and is taking over the election
- Eventually, highest-numbered running process is the only one left standing
  - Announces results of election to all others

Bully: Example



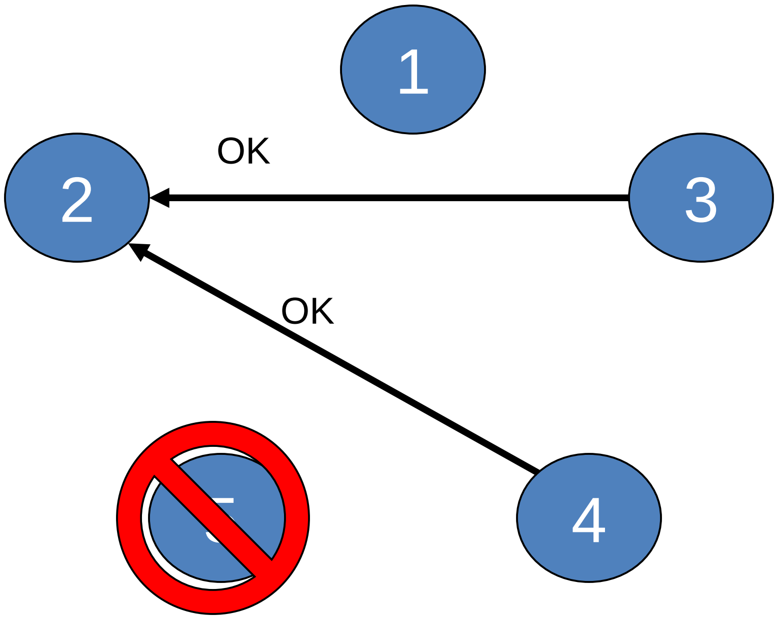
- Process 5 is the current coordinator
- Process 5 crashes
- Process 2 notices it first

Bully: Example



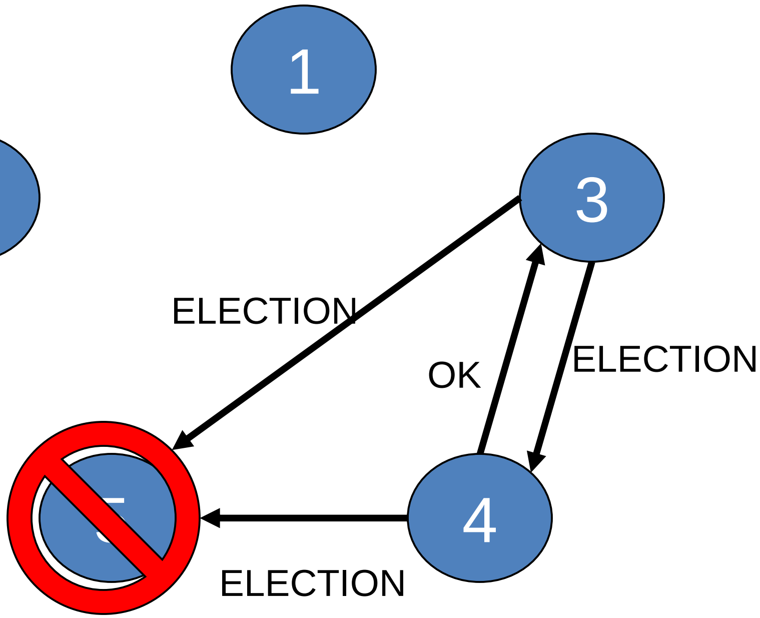
- Process 2 starts election
  - Sends ELECTION to all higher processes (= 3, 4, 5)

Bully: Example



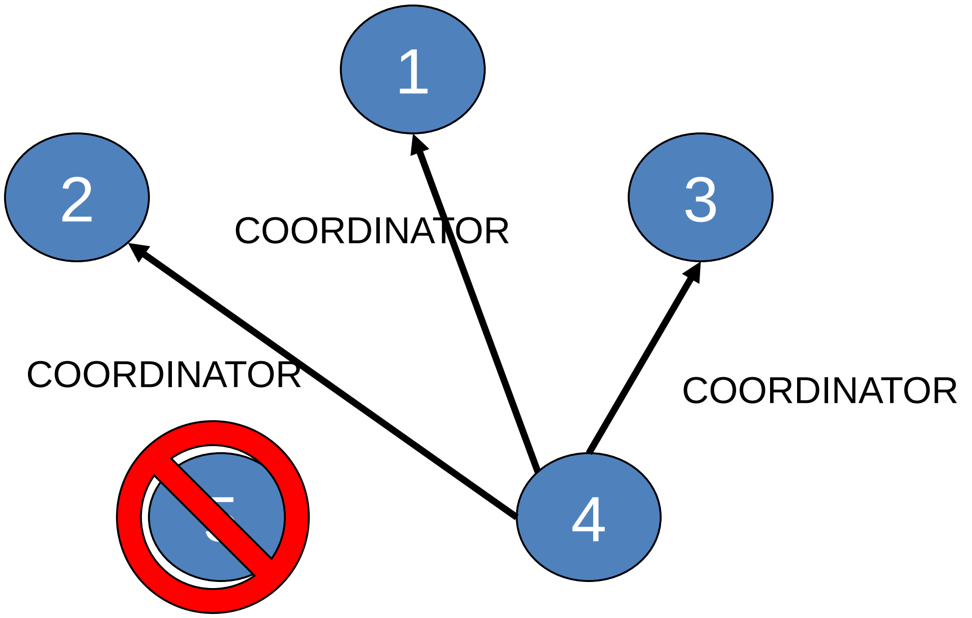
- Processes 3 and 4 reply to 2 with OK
- Process 2 has done its job, now it just waits for new coordinator

Bully: Example



- Processes 3 and 4 start elections by sending ELECTION to higher numbered processes
- Process 4 sends OK to process 3

Bully: Example



- Process 4 notices that process 5 is dead, knows that it is now the coordinator
- Process 4 sends COORDINATOR to all other processes

Bully: Remarks

- Need to know maximum bound for message delivery
  - Usually implies synchronous system
- Name comes from “biggest guy taking over”
  - Process with highest number bullies others into accepting it as the coordinator
- Every process that receives ELECTION will start its own election
  - Seems redundant, but eventual winner may crash during algorithm
  - Note: Algorithm works even if processes crash during election
- Best case performance: Process with highest number running detects crash
  - No election, just COORDINATOR messages

Election on Rings

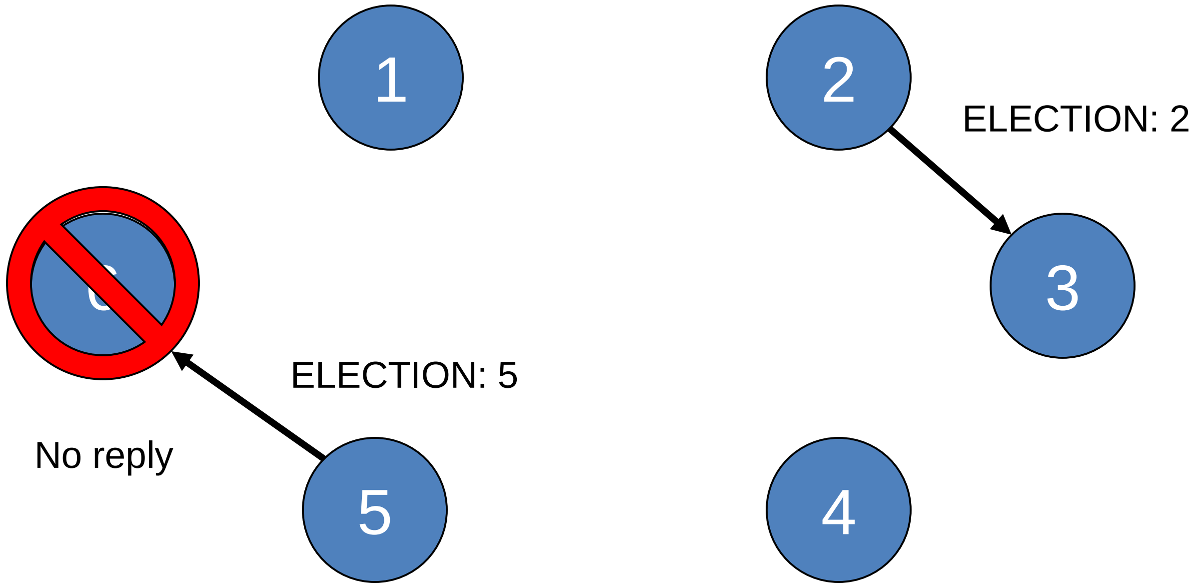
Processes arranged in a ring

- Each process knows its successor

Algorithm works as follows:

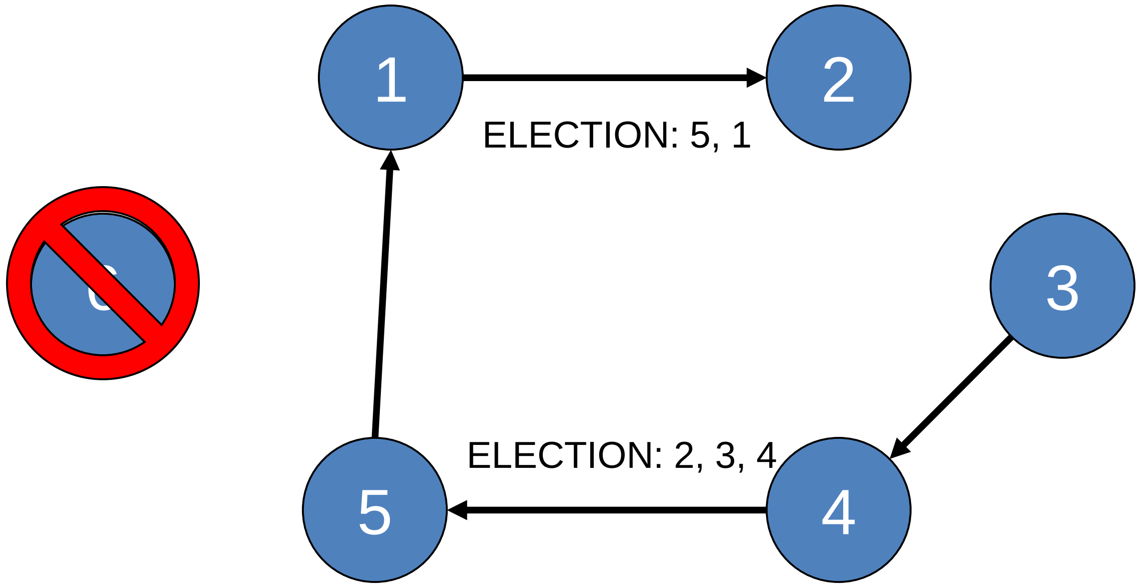
1. If a process notices coordinator is down, it builds ELECTION message with its own process number
   - Send message to successor
   - If successor is down, then to successor's successor, etc. until a running process is found
2. At every step, processes add their own numbers to message
3. When message comes back to initiator, change type to COORDINATOR and circulate it again
   - Knows this when received message has its own number
4. When COORDINATOR has gone around, it is removed
   - At this point all processes know new coordinator

Election on Rings: Example



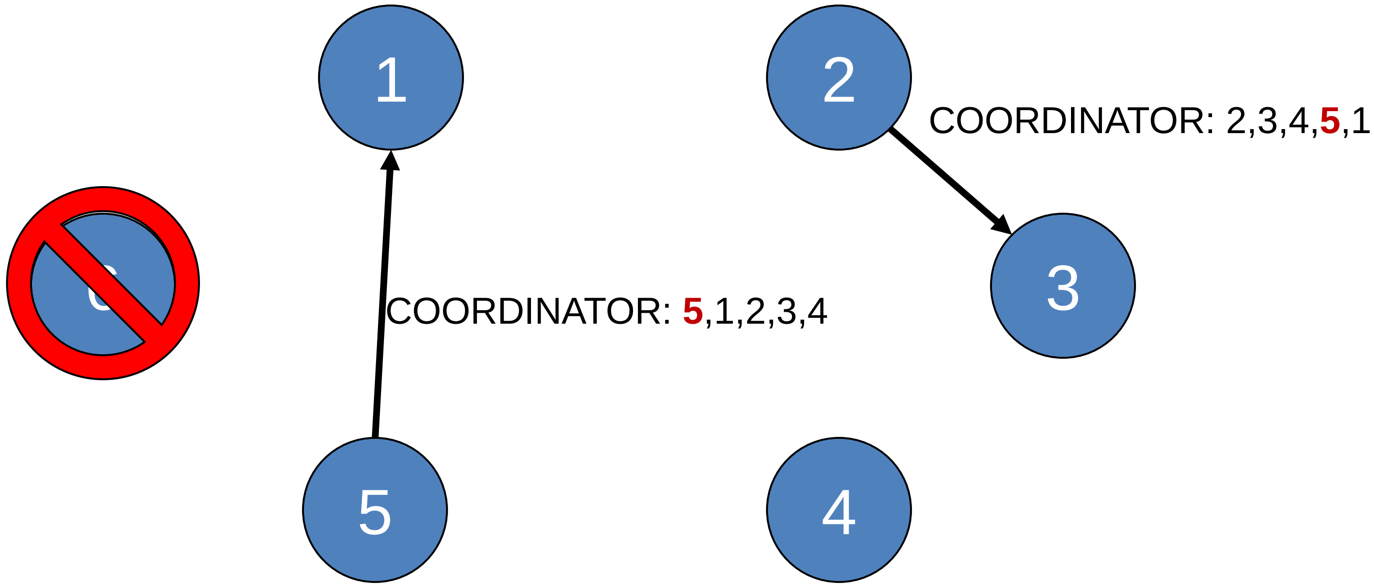
- Process 6 was coordinator, but has crashed
- Processes 2 and 5 happen to notice it at the same time
- Both send ELECTION to successors

Election on Rings: Example



- ELECTION messages are circulated
- Eventually both messages reach their senders
  - Messages have collected information about all the other processes in the ring

Election on Rings: Example



- When messages reach senders, they send COORDINATOR messages with list of all processes
- COORDINATOR messages also go all the way around the ring
- Multiple COORDINATOR messages do no harm

Properties & Performance: Election

- desired properties
  - E1: once participating, proc p_i has $elected_i = \perp$ (undef.) OR $elected_i = P$ where P is proc to be chosen at end of run with largest ID (safety)
  - E2: all (non crashing) proc‘s p_i will eventually set $elected_i \neq \perp$ (liveness)
- performance
  - net bandwidth utilization (proportional to total # of msgs sent)
  - turnaround time: # of serialized msg Xmission times between initiation and termination of single run
- further issues:
  - (which) failures tolerated?
  - concurrent execution tolerated?
  - synchronous vs. asynchronous systems?