Distributed Systems

ID2201

coordination

Johan Montelius

Coordination

- Coordinating several threads in one node is a problem, coordination in a network is of course worse:
  - failure of nodes and networks
  - no fixed coordinator
  - no shared memory
- Coordination is often the problem of:
  - deciding who is to decide
  - knowing who is alive.

Fundamental models

- Interaction model:
  - Is the system asynchronous or synchronous?
  - Can we assume a node has crashed if it does not reply?
- Failure model:
  - Will nodes crash?
  - Will crash nodes return to life?
  - Is crashing the only failure?

Failure detectors

- How do we detect that a process has crashed and how reliable can the result be?
  - unreliable: result in *unsuspected* or *suspected* failure
  - reliable: result in *unsuspected* or *failed*
- Reliable detectors are only possible in synchronous systems.

Distributed algorithms

- We will look at some distributed algorithms and consider:
  - reliable systems: if nothing goes wrong
  - unreliable systems: but nodes fail by crashing and this can be detected by reliable failure detectors

Three sides of the same coin

- Mutual exclusion
  - Decide who is to enter a critical section.
- Leader election
  - Decide who is to be the new leader.
- Atomic multicast

  Which messages, and in which order, should be deliverd.

Distributed mutual exclusion

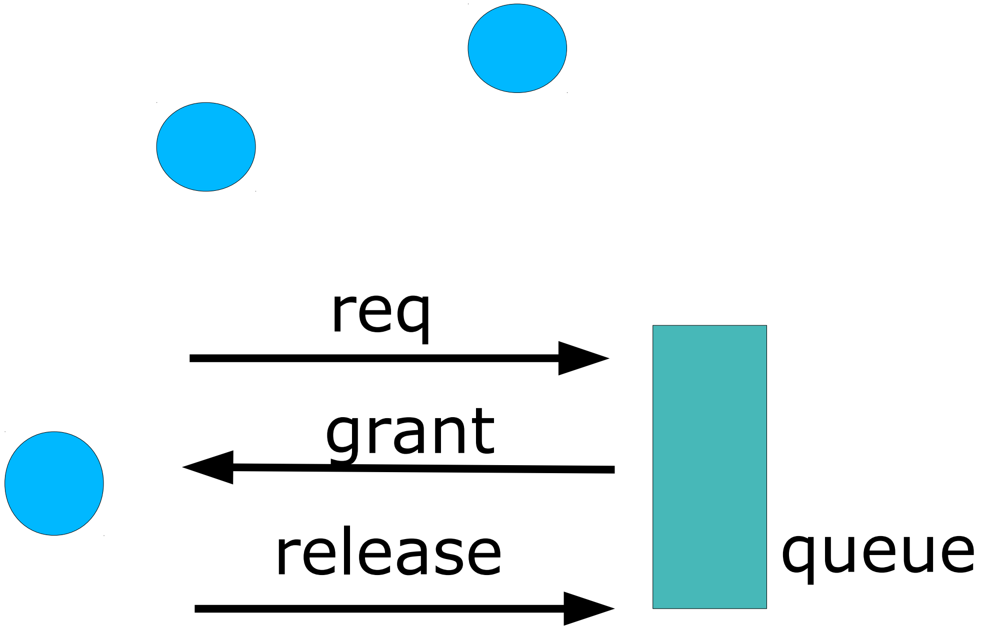
- Requirements
  - **Safety:** at most one process may be in critical section at a time
  - **Liveness:** *starvation free,* deadlock free
  - **Ordering:** allowed to enter in *request happened-before* order

Evaluation

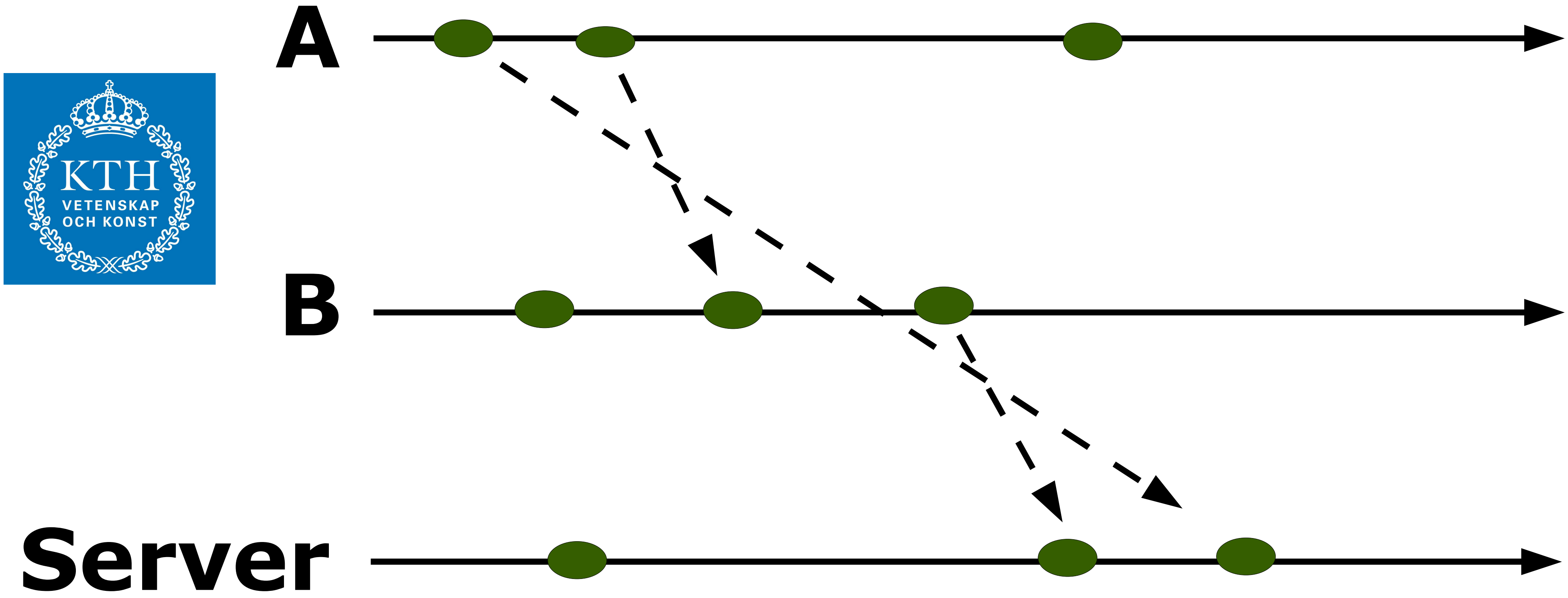
- Number of messages needed.
- Client delay:
  - worst,
  - mean or, average time to enter critical section
- Synchronization delay: how long time between exit and enter.

Central service algorithm

- Requirements?
  - safety
  - liveness
  - ordering



Ordering - what is a request



Performance

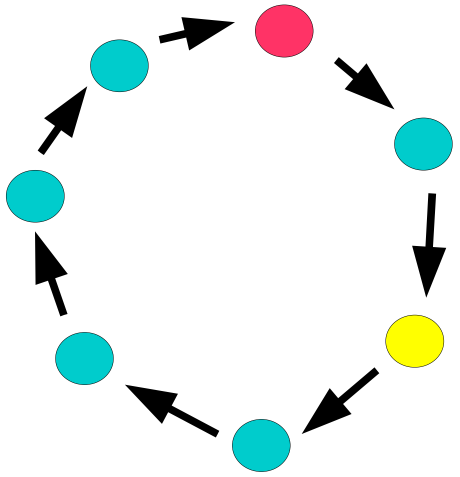
- messages
  - enter: request, grant
  - exit: release
- client delay
  - enter: message round trip plus waiting in queue
  - exit: constant (asynchronous message)
- synchronization delay
  - round trip: release - grant

Failure

- What can happens if we allow nodes to fail?
  - a client
  - a client holding the token
  - the server
- What if we have reliable failure detectors?
- Can we do with unreliable failure detectors?

Ring-based algorithm

- Requirements
  - safety
  - liveness
  - ordering

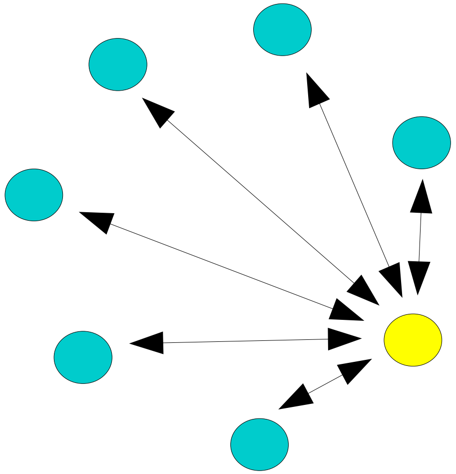


Ring-based algorithm

- Performance
  - messages
  - client delay
  - synchronization delay
- Failure
  - the lost token

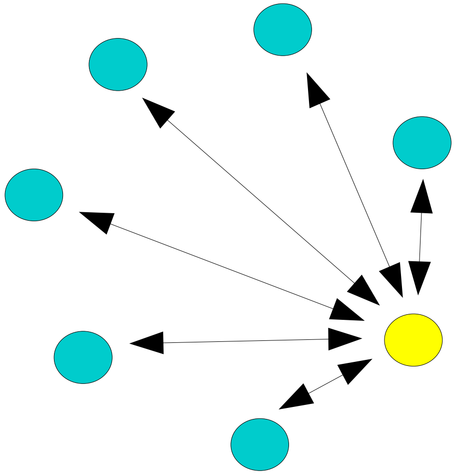
Distributed algorithm

- Send request to all peers.
- When all peers have acknowledged the request, enter the critical section.
- What could go wrong?



Distributed algorithm

- Break deadlock
  - introduce priority
- Fairness
  - Ricart and Agrawala



Ricart and Agrawala

- Enter:
  - enter state waiting and broadcast a request {T,i} containing a *Lamport time stamp T* and process id I to all peers
  - wait for replies from all peers
  - enter state held
- Receiving a request {R,j}:
  - if *held or (waiting* and {T,i} < {R,j}) then queue request, else reply ok
- Exit:
  - reply to all queued requests

Ricart and Agrawala

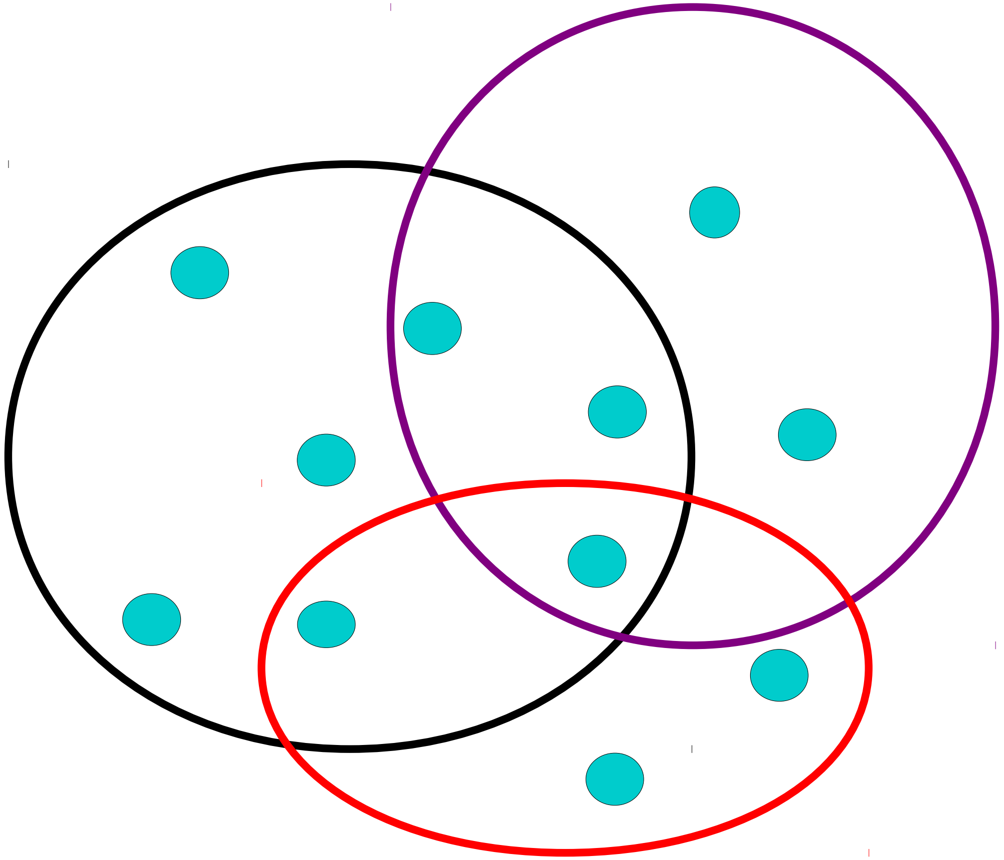
- Requirements
  - safety, liveness, ordering
- Efficiency
  - messages
  - client delay
  - synchronization delay
- Failure
  - not so good

Maekawa's voting

- Why have permission from all peers, it's sufficient to have votes from a subset S if no one can enter with the votes from the complement of S.
- The subset S is called a *quorum.*

Maekawa's voting

- Requirements
  - safety
  - liveness
  - ordering



Maekawa's voting

- Efficiency
  - messages: twice sqrt(N)
  - client delay: round trip
  - synchronization delay: one message
- Failure
  - not that bad?

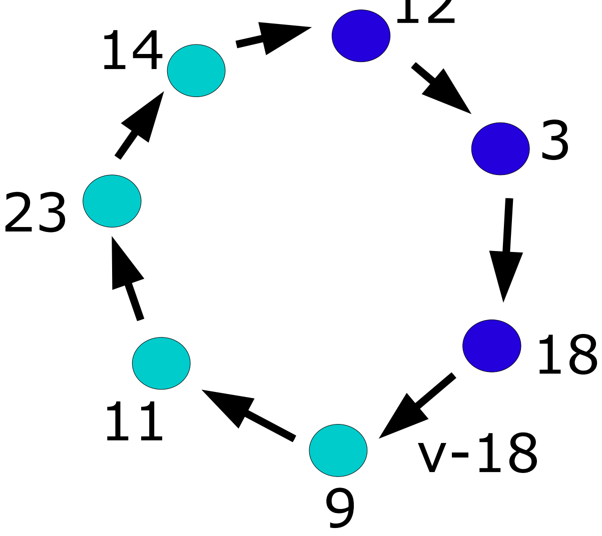
Election

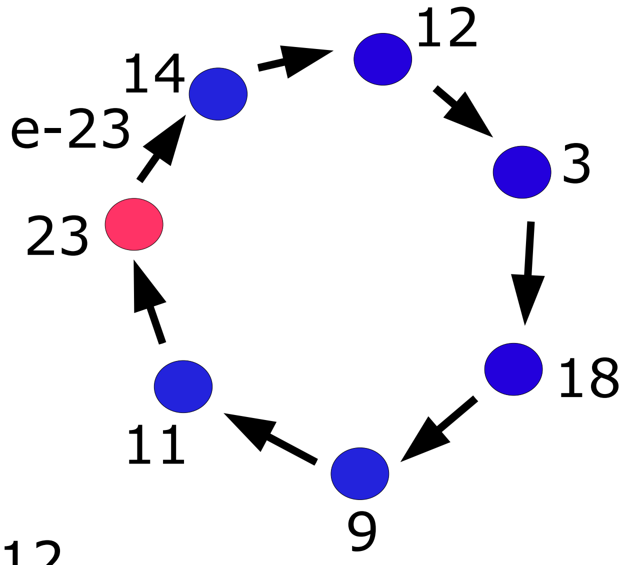
- Many algorithms require a server but if no node is assigned to be the server or if the server crashes we need to find a new server.
- Assumptions:
  - any node can *call an election* but it can only call one at a time
  - a node is either *participant* or *non-participant*
  - nodes have identifiers that are ordered

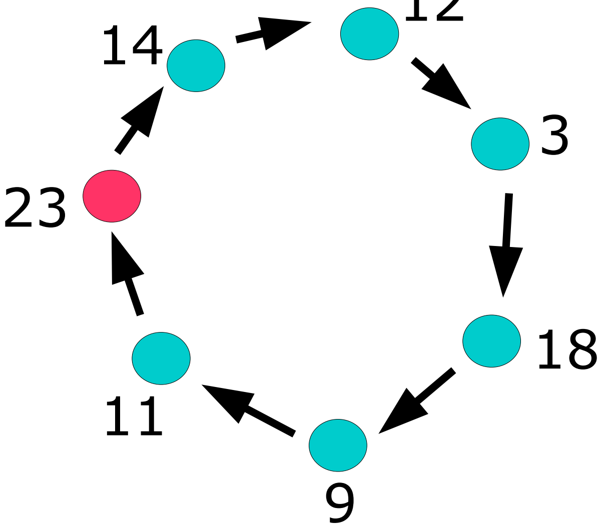
Election

- Requirements
  - safety: a *participant* is either non-decided or decided with P, a unique non crashed node
  - liveness: all nodes eventually *participate* and decide on a elected node
- Efficiency
  - number of messages
  - turnaround time: delay from *call* to *close*

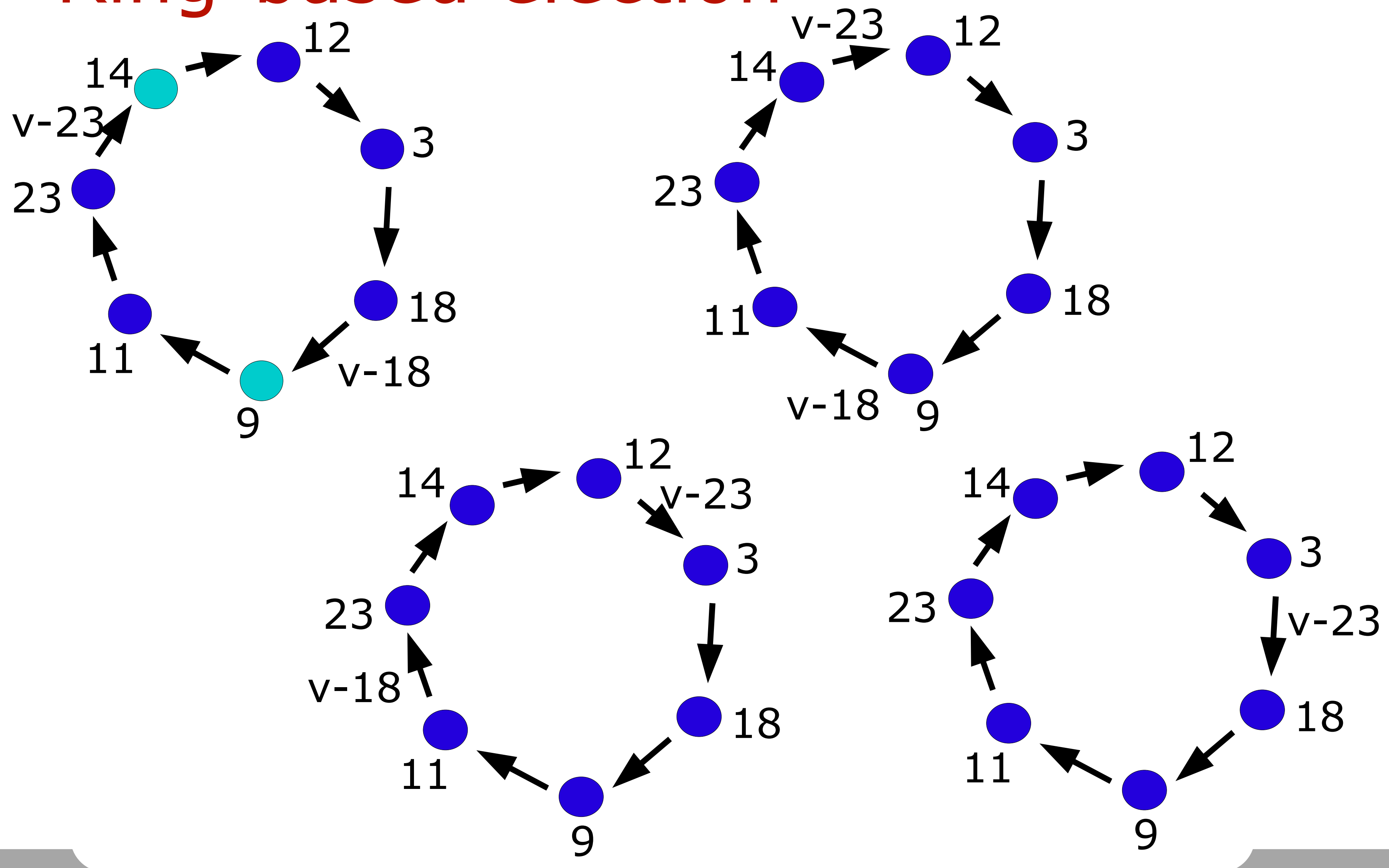
Ring-based election







Ring-based election



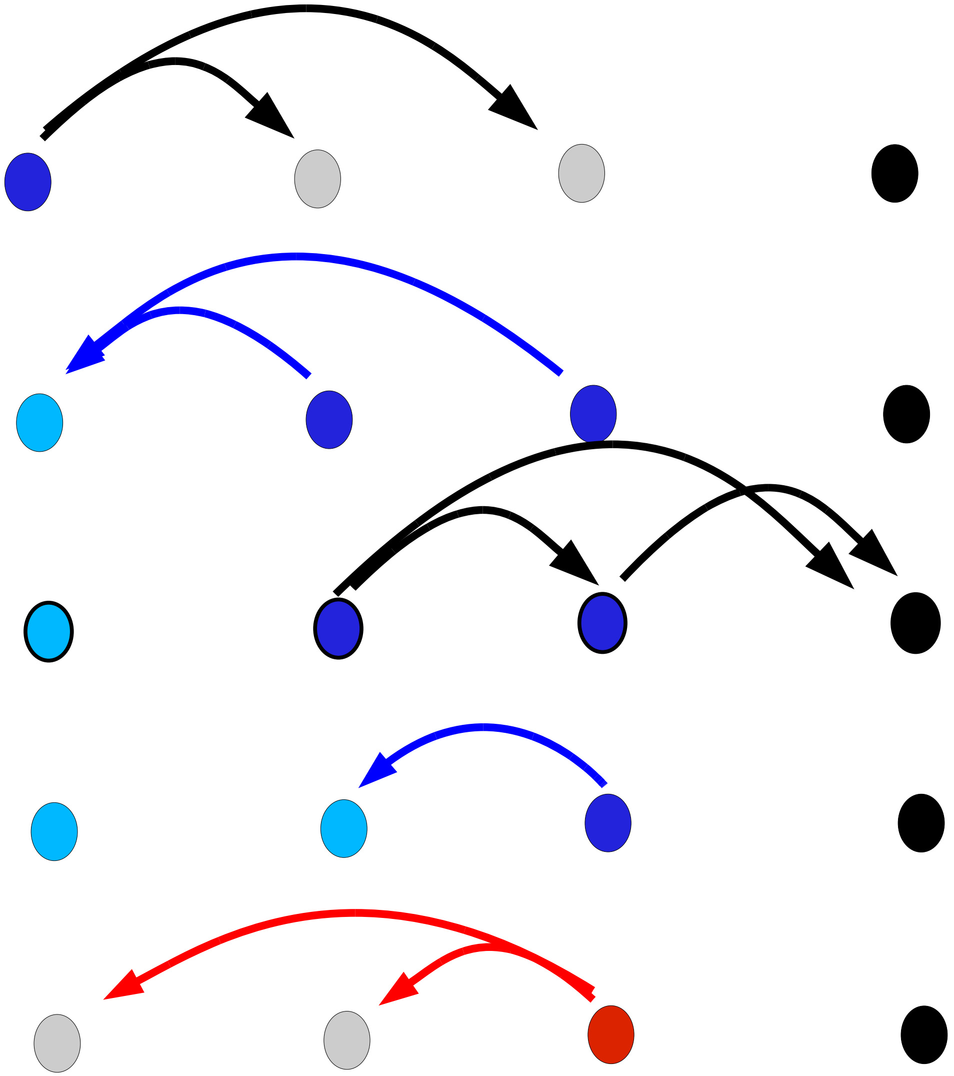
Ring-based election

- Requirements
  - safety
  - liveness
- Efficiency
  - messages: best case, worst case?
  - turnaround:
- Failure
  - hmm, ...

The bully algorithm

- Nodes have identifiers and are ordered.
- Any node can reliably send messages to any other higher node.
- Nodes can crash (and remain dead) and this is *reliably* detected.
- Algorithm starts when a node detects that *the coordinator* has crashed.

The bully algorithm



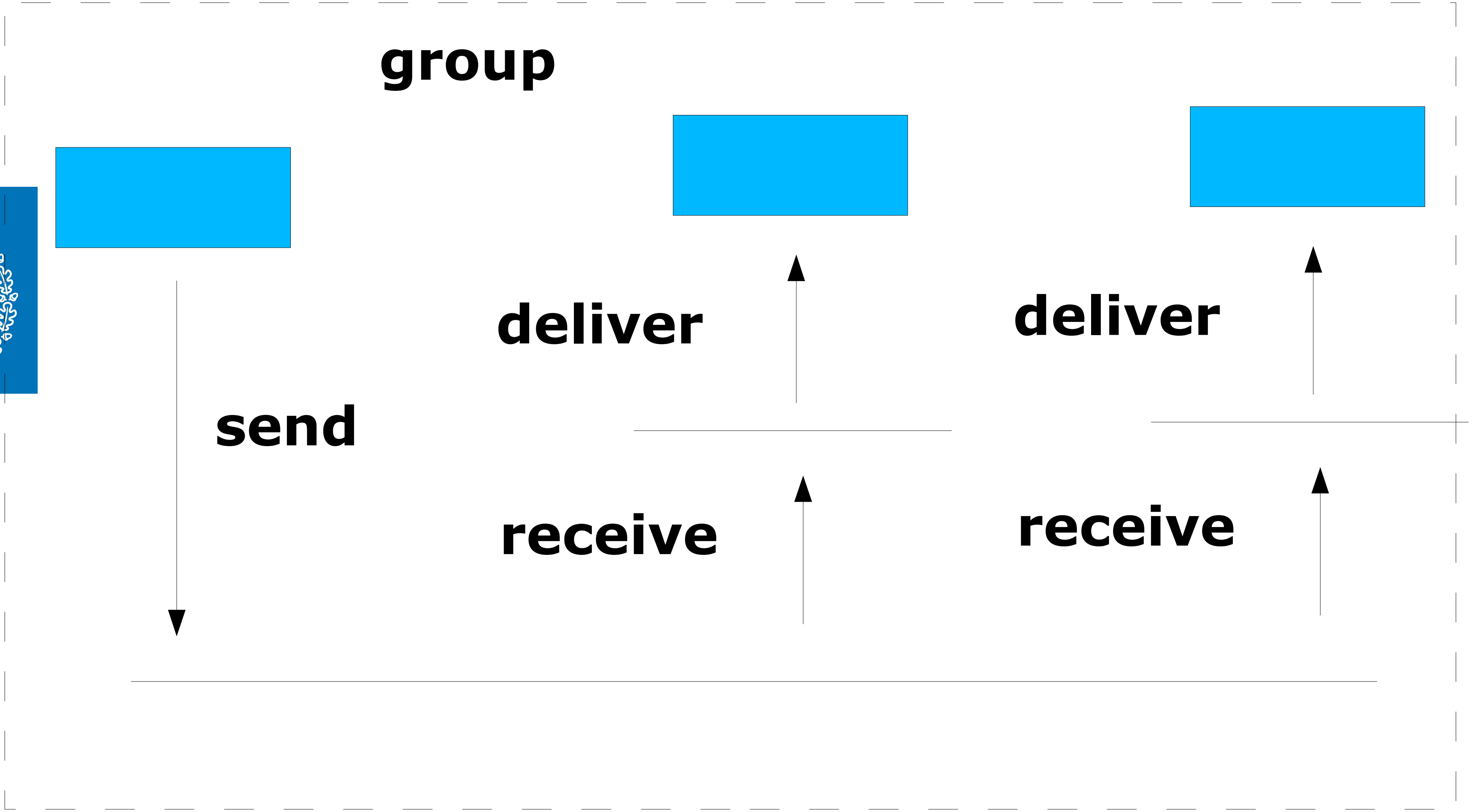
The bully algorithm

- Requirements
  - safety ... hmm
  - liveness
- Efficiency
  - Messages:
    - best case, worst case
  - Turnaround:

Multicast communication

- Multicast:
  - Sending a message to a specified group of n nodes.
- Reliable multicast:
  - All nodes see the same messages.
- Atomic multicast:
  - All nodes see the same messages in the same order.

Model



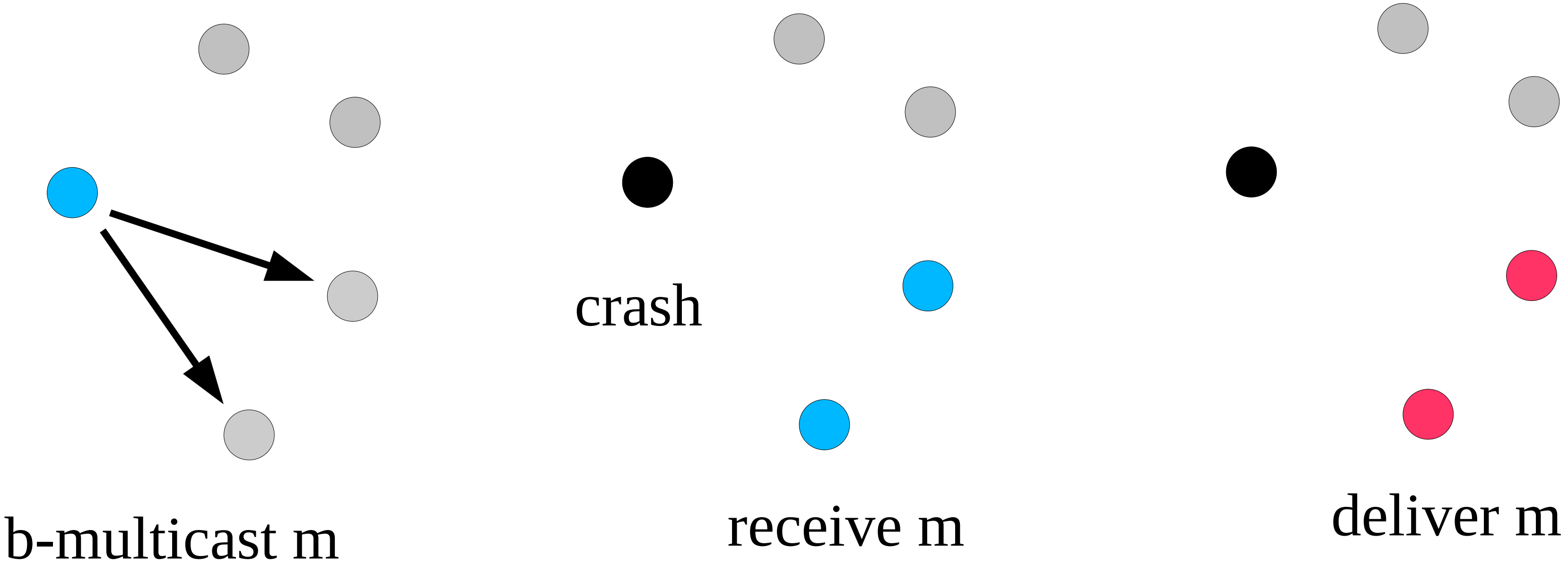
Requirements

- Integrity
  - a process *delivers* a message at most once and only deliver messages that have been sent
- Validity
  - if a process multicast m then it will also eventually deliver m
- Agreement
  - if a process *delivers m* then all processes in the group eventually *delivers m*

Basic multicast

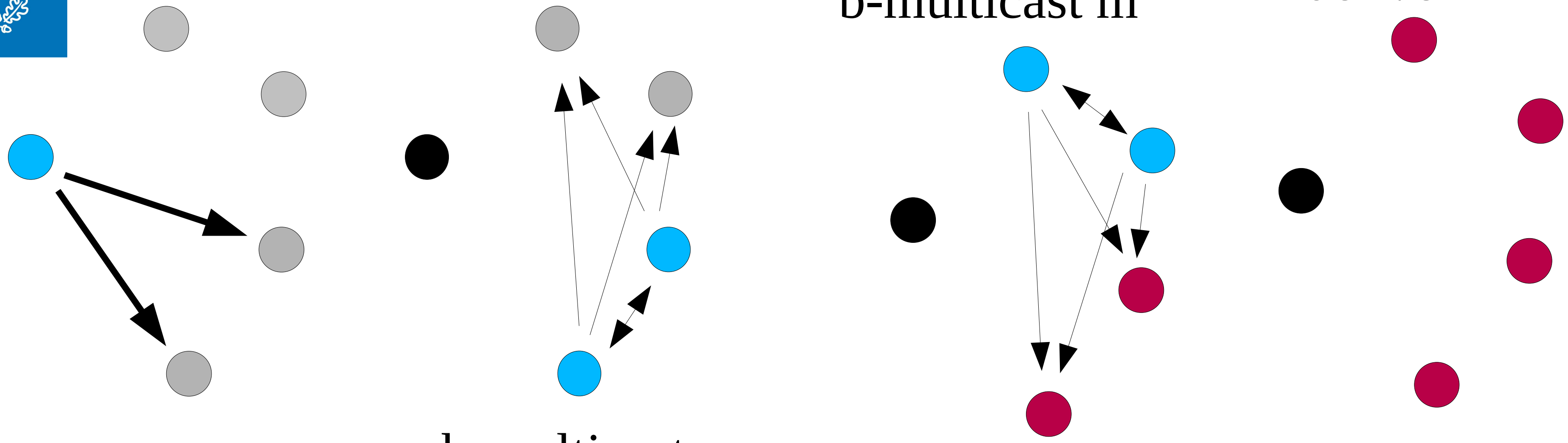
- To b-multicast a message m:
  - send m to each process p
- If m is received:
  - b-deliver m
- What was the problem?

Basic multicast



Reliable multicast

- Can we implement reliable (atomic) multicast if the only thing we have is basic multicast?



b-multicast m

b-multicast m

b-multicast m

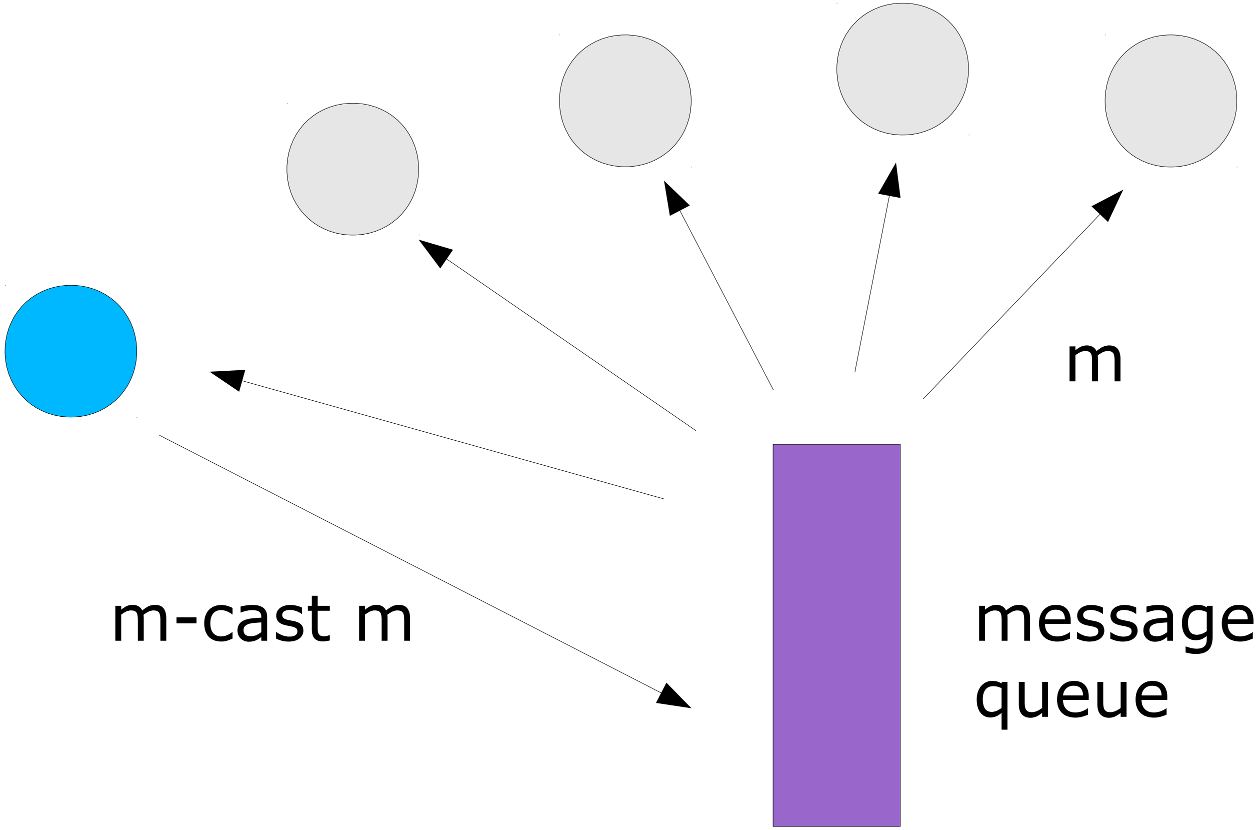
r-deliver m

r-deliver m

Ordered multicast

- The problem with the reliable multicast is that multicast messages might arrive in different order at different nodes.
- Requirements:
    - FIFO order: delivered in order as sent by the sender
    - Causal order: delivered in order as *happened before* sent order
    - Total order: delivered in *same order* by all processes

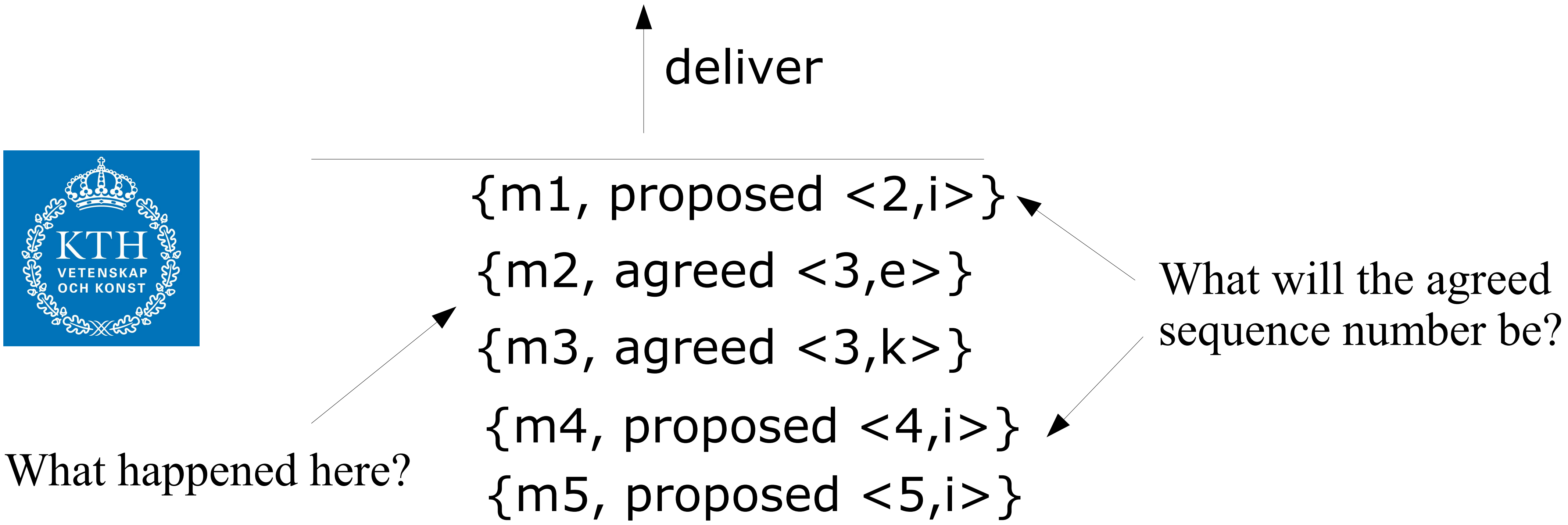
Sequencer



Distributed - ISIS

- Multicast a message and request a sequence number.
- When receiving a message, *propose a sequence number (including process id)* and place in an ordered hold-back queue.
- After collecting all proposals, select the highest and multicast agreement.
- When receiving agreement tag message as agreed and reorder hold-back queue.
- If first message in queue is decided then deliver.

the hold-back queue



deliver

{m1, proposed <2,i>}

{m2, agreed <3,e>}

{m3, agreed <3,k>}

{m4, proposed <4,i>}

{m5, proposed <5,i>}

What happened here?

What will the agreed sequence number be?

Causal ordering

- How can we implement casual ordering?
  - multicast vector clock holds number of multicast operations
  - tag each multicast message with multicast clock
  - hold b-delivered messages until clock of message is *less* (modulo sender) than own current message clock
  - update own message clock
- Only multicasted messages are counted.

Summary

- Coordination in distributed systems is problematic.
- If we have a fixed set of nodes and can detect failures there are many solutions.
- Three sides of the same coin:
  - mutual exclusion
  - leader election
  - atomic multicast