



# Programming of Distributed Systems

Topic V – Coordination

Dr.-Ing. Dipl.-Inf. Erik Schaffernicht

# Reading Remarks

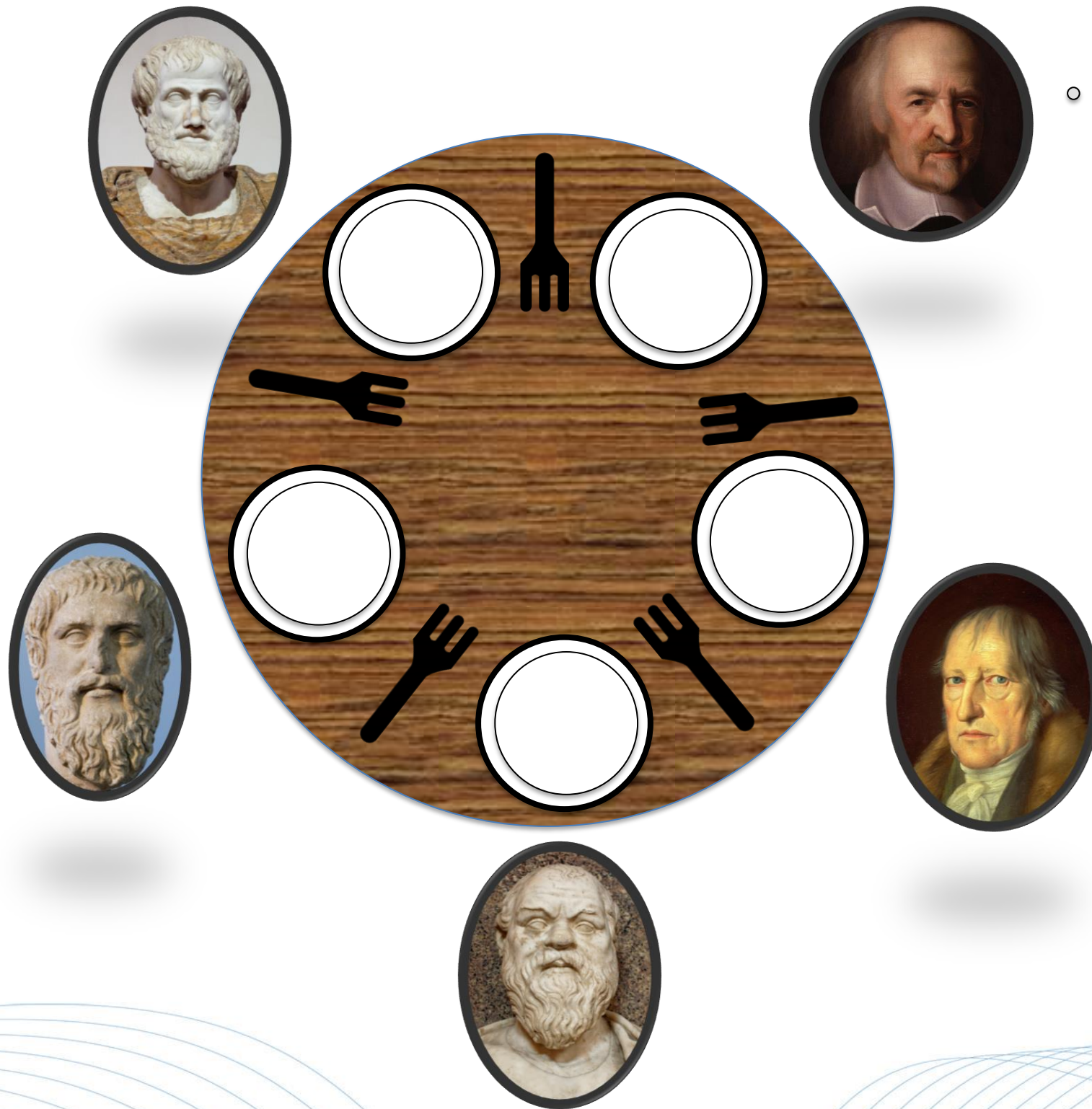
**Reading Task:**

Chapter 6.3 & 6.4

*Skip:* Chapter 6.5, 6.6 & 6.7

Chapter 8.1, 8.2 (skip PAXOS) & 8.6 (skip 8.4 & 8.5)

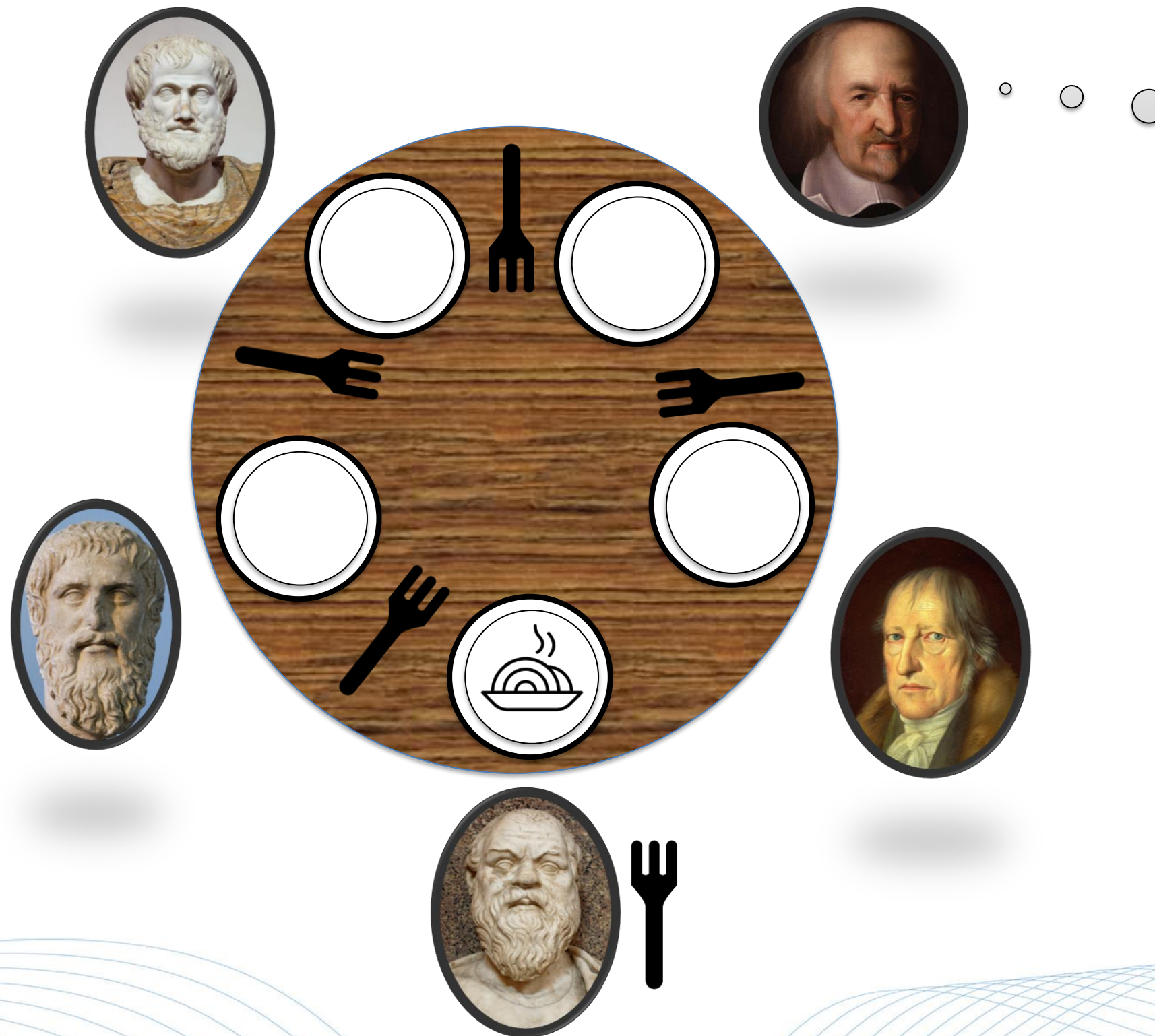
# Dining Philosophers



```
forever do {  
  think()  
  eat:  
    take_right_fork()  
    take_left_fork()  
    gormandize()  
    return_right_fork()  
    return_left_fork()  
}
```

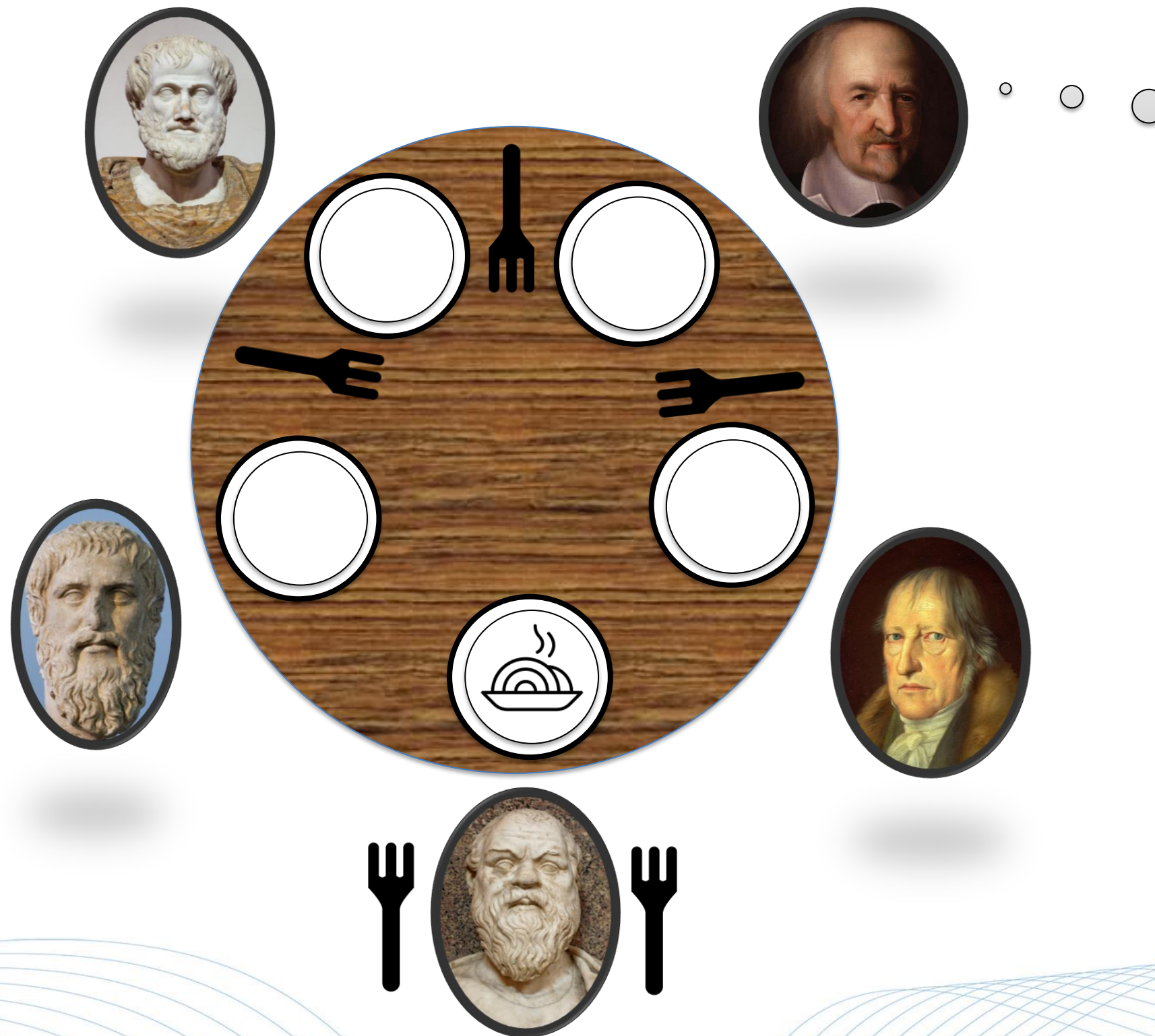


# Dining Philosophers



```
forever do {  
  think()  
  eat:  
    take_right_fork()  
    take_left_fork()  
    gormandize()  
    return_right_fork()  
    return_left_fork()  
}
```

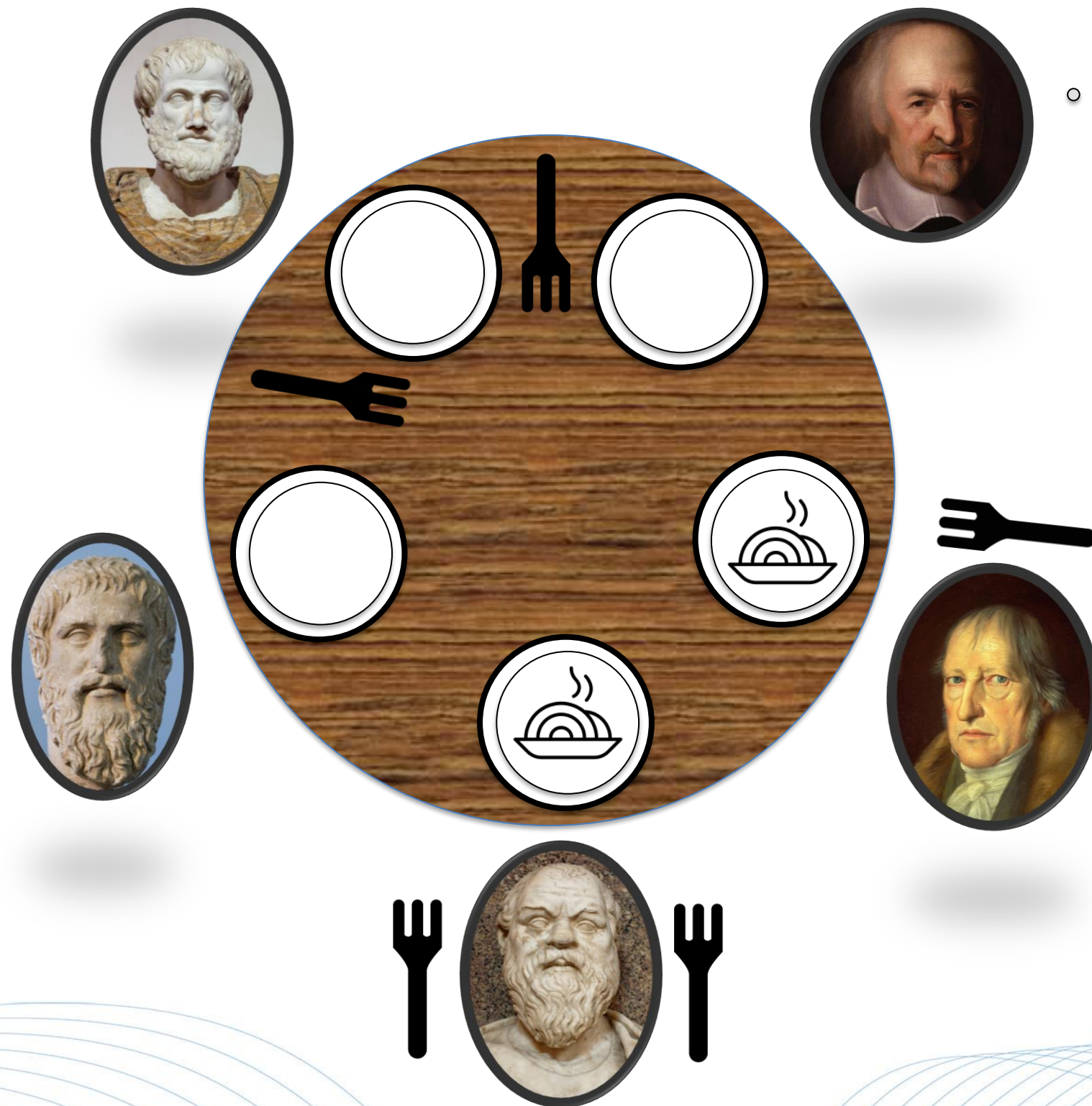
# Dining Philosophers



```
forever do {  
  think()  
  eat:  
    take_right_fork()  
    take_left_fork()  
    gormandize()  
    return_right_fork()  
    return_left_fork()  
}
```



# Dining Philosophers

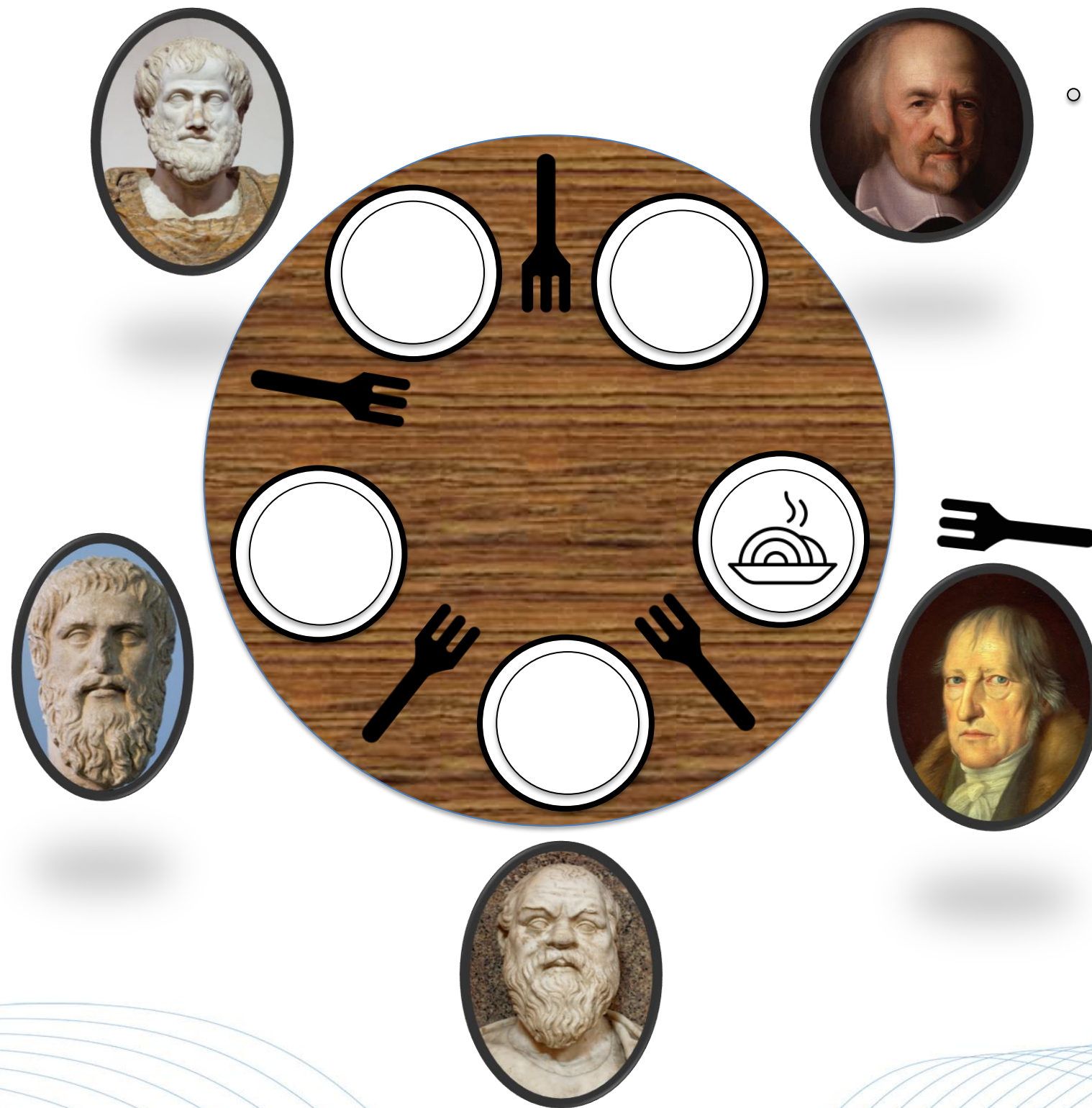


```
forever do {  
  think()  
  eat:  
    take_right_fork()  
    take_left_fork()  
    gormandize()  
    return_right_fork()  
    return_left_fork()  
}
```





# Dining Philosophers

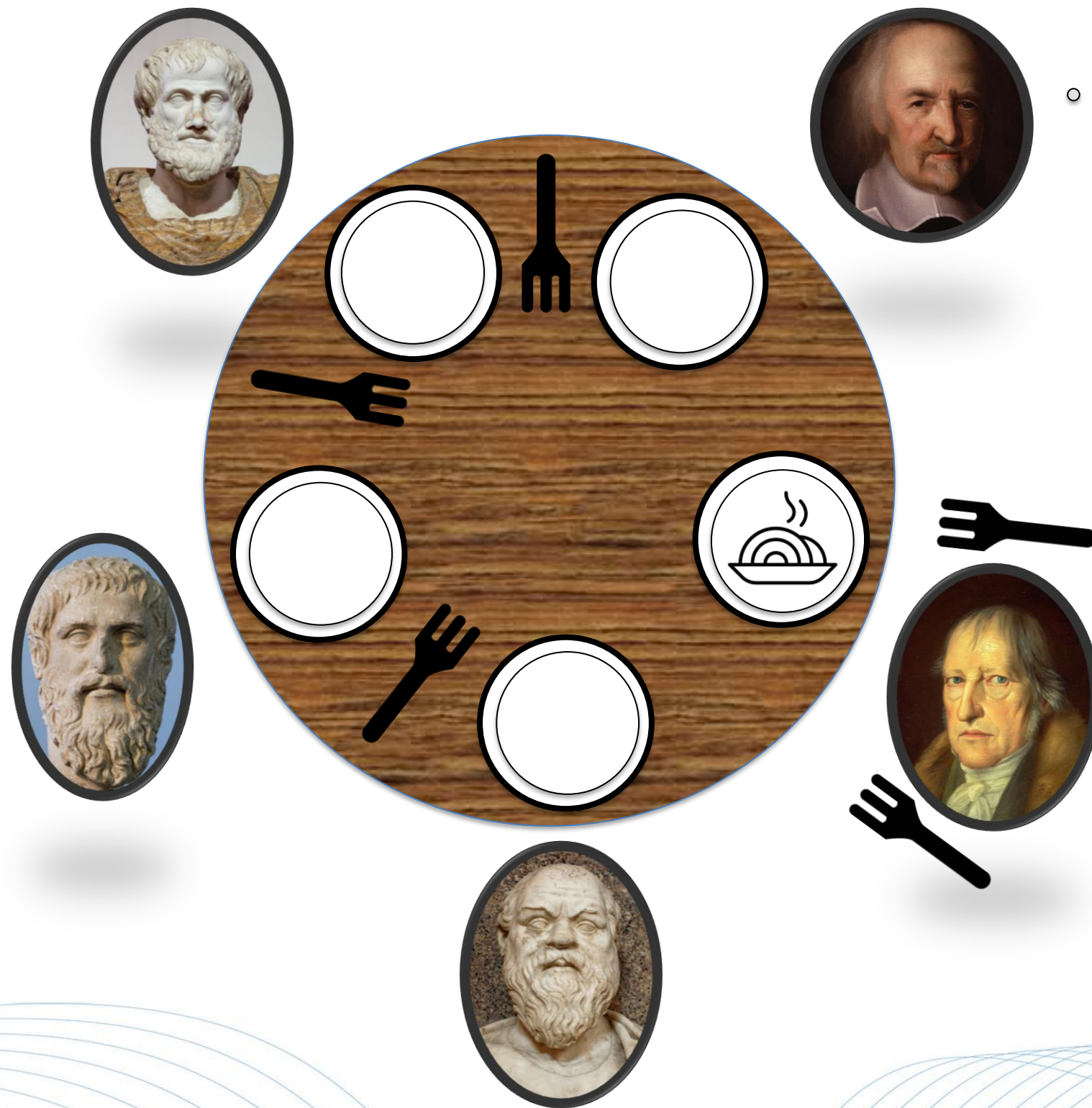


```
forever do {  
  think()  
  eat:  
    take_right_fork()  
    take_left_fork()  
    gormandize()  
    return_right_fork()  
    return_left_fork()  
}
```





# Dining Philosophers

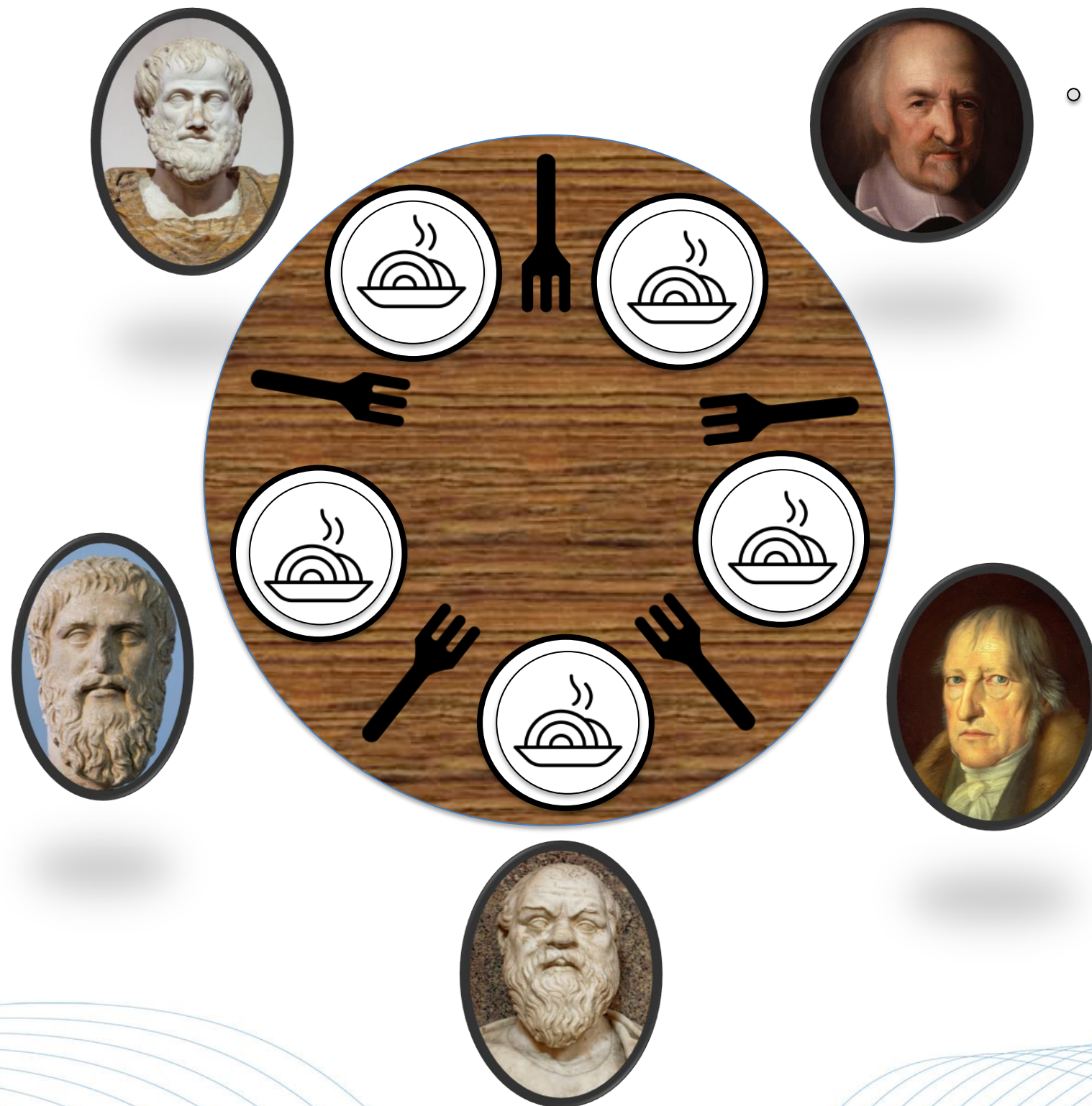


```
forever do {  
  think()  
  eat:  
    take_right_fork()  
    take_left_fork()  
    gormandize()  
    return_right_fork()  
    return_left_fork()  
}
```





# Dining Philosophers



```
forever do {  
  think()  
  eat:  
    take_right_fork()  
    take_left_fork()  
    gormandize()  
    return_right_fork()  
    return_left_fork()  
}
```



# Dining Philosophers



```
forever do {  
  think()  
  eat:  
    take_right_fork()  
    take_left_fork()  
    gormandize()  
    return_right_fork()  
    return_left_fork()  
}
```



# Dining Philosophers



```
forever do {  
  think()  
  eat:  
    take_right_fork()  
    take_left_fork()  
    gormandize()  
    return_right_fork()  
    return_left_fork()  
}
```

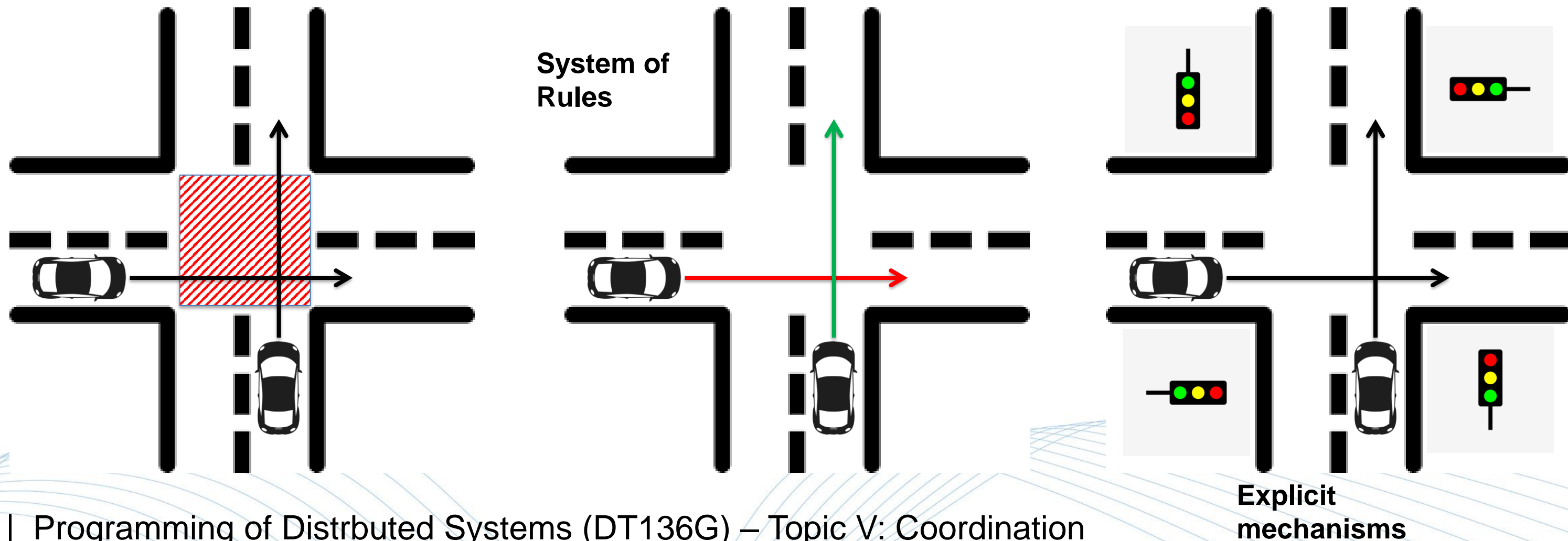
# Non-functional algorithm properties

- deadlocks
- starvation
- fairness
- robustness
- ressource requirements
- scalability



# Critical Sections in Distributed Systems

A phase in which a node is accessing exclusively a resource is called **critical section (CS)**. These sections require **mutual exclusion** of concurrent nodes.



# System of rules

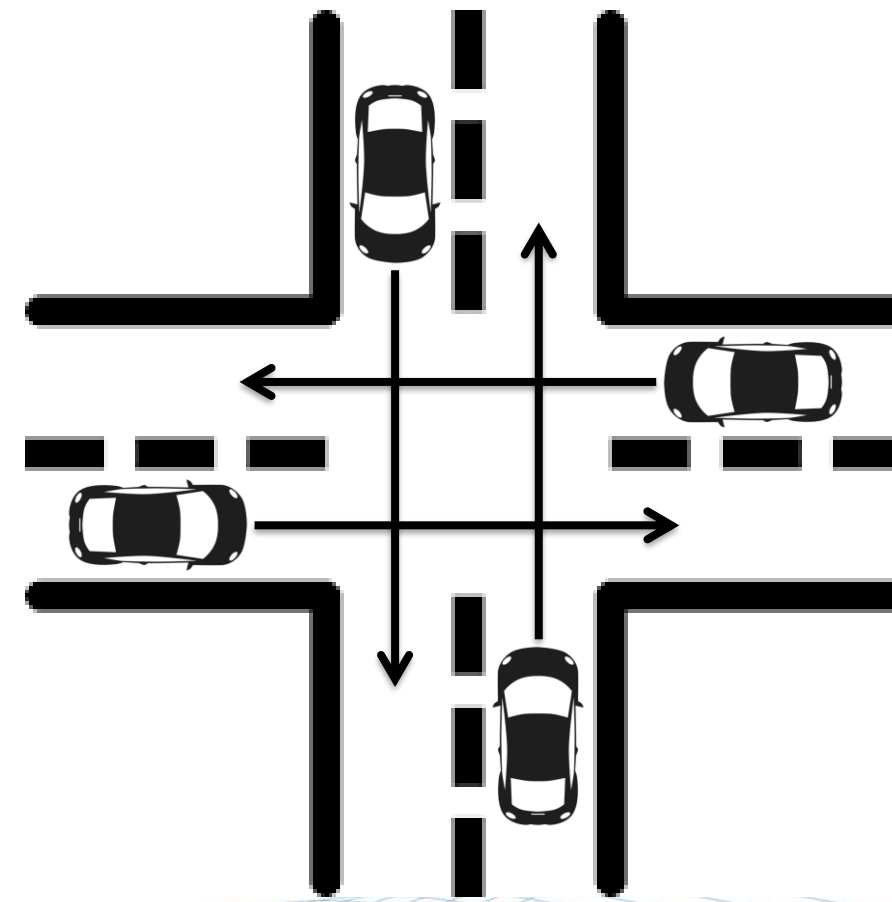
Simple rule systems are not always complete or well designed.

→ dead-locks

## Example

”right-before-left”-rule

- cheap and simple
- not dead-lock free





# Distributed Algorithms for Mutual Exclusion

## Problem

Lack of hardware / operating system support to implement semaphores or monitors

## Requirements

- **correctness** – only one\* node is using the critical section
- **liveness** – no deadlocks and no starvation of nodes
- **fairness** – access sequence for a critical section follows an order

# Central Coordination

## Idea

Adaption of the methods of non-distributed systems

≈ semaphore

- all competitors agree on a coordinator
- before entering a CS
  - sending an application to the coordinator (`enter-message`)
  - waiting for the go-ahead from the coordinator (`grant-message`)
- on leaving a CS informing the coordinator (`leave-message`)

### Client:

```
send(coordinator, enter)
receive(grant)
```

```
<operations in the
critical section>
```

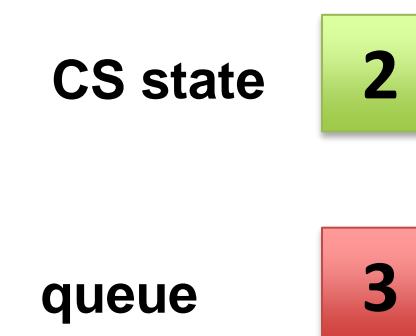
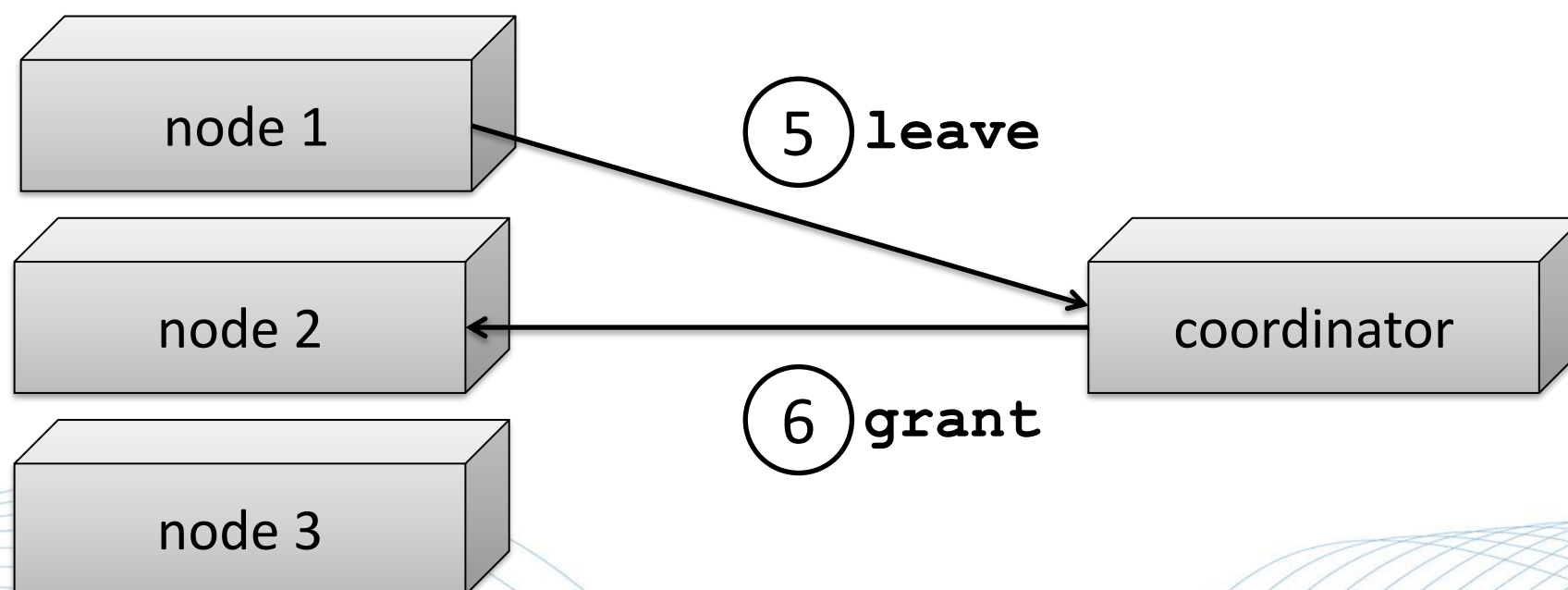
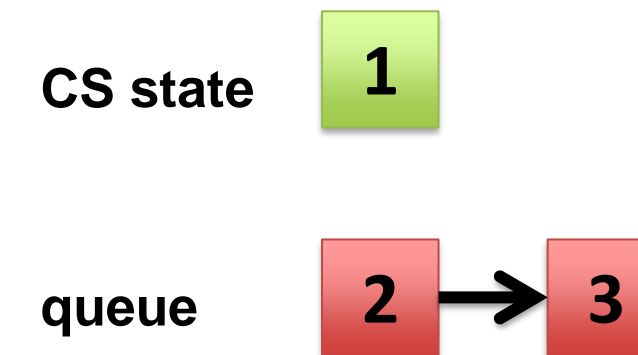
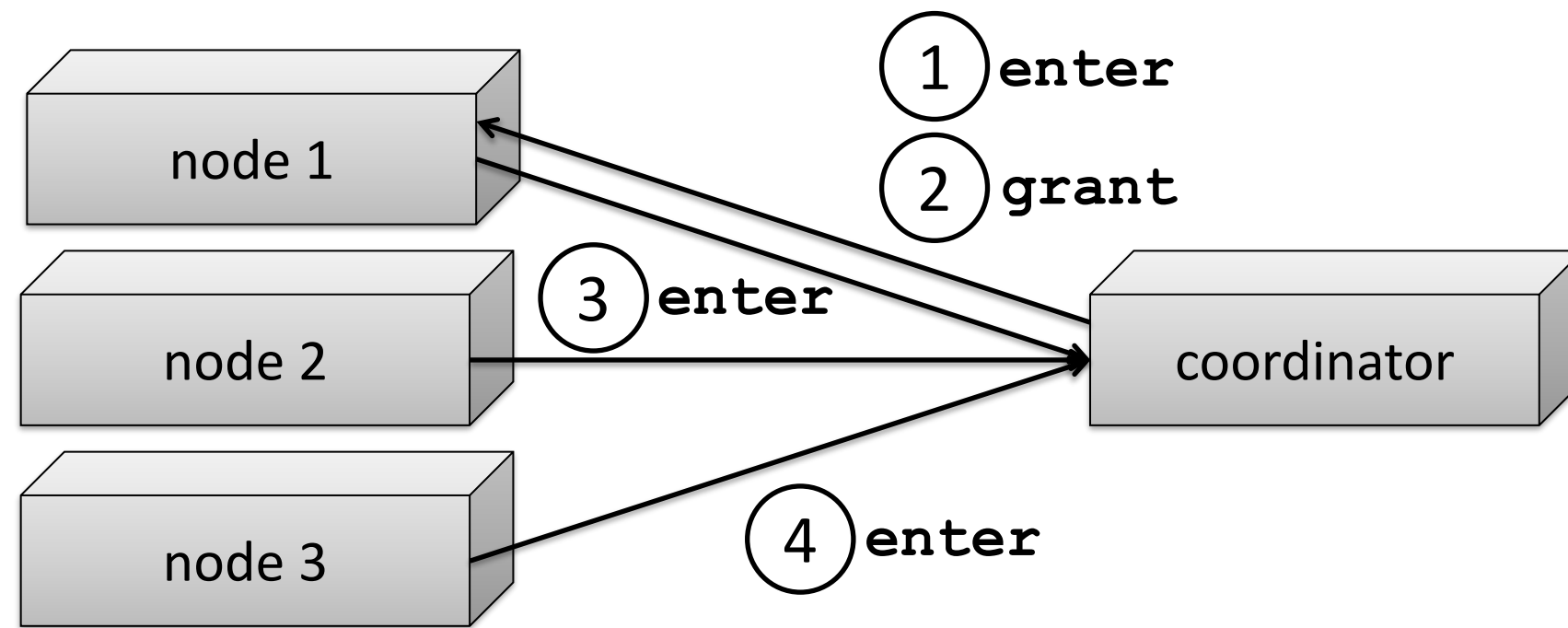
```
send(coordinator, leave)
```



# Coordinator

- Implements a sequence strategy (e.g. FCFS, priority-based)
- For each CS the coordinator records
  - current state of the CS (mutex-like)
  - an application queue
- responds to `enter`
  - with `grant` if CS is available
  - else the caller is added to the application queue
- responds to `leave`
  - with `grant` to one node in the application queue

# Central Coordination





# Central Coordination

## Pros

- easy to implement
- **correctness** is easy to verify
- **liveness**
  - Coordinator: for each `enter` there will be (at some time) a `grant`
  - Clients: for each `grant` there will be (at some time) a `leave`
- **fairness** options (→ flexible)
  - random choice from the queue
  - order (tpco) or timestamp (sync time) of sending `enter` messages
  - order of receiving `enter` messages (ppco) or priority based

# Central Coordination

## Pros

- Complexity
  - time: 3 operations per CS access  $\rightarrow O(1)$
  - communication message count: 3 per CS access  $\rightarrow O(1)$
  - communication message volume: 3 message types  $\rightarrow O(1)$

## Hmmm...

- Scalability in regard to the number  $b$  of CS accesses
  - time, message count and volume  $\rightarrow O(b)$
- $\rightarrow$  independent of client number, linear in access requests
- $\rightarrow$  coordinator involved in everything  $\rightarrow$  bottleneck



# Central Coordination

## Cons

- Fail-stop failures
  - coordinator → gradually blocks all clients, single-point-of-failure  
→ liveness
  - client → might block CS indefinitely  
→ liveness
- message loss
  - enter → blocks client
  - leave, grant → blocks CS indefinitely  
→ liveness

# Central Coordination

## Cons

- Byzantian failures
  - coordinator (lies)  
→ correctness, liveness, fairness
  - client (doesn't send `enter/leave` or doesn't wait for `grant`)  
→ correctness, liveness, fairness
  - message changes  
→ correctness, liveness, fairness



# Central Coordination

## Summary

- Fast, simple, few messages
- coordinator is bottleneck
- zero tolerance for failures

Reason for the problems is ignoring

- virtues of distributed systems: parallelism, load sharing
- challenges of distributed systems: partial failures

# Decentralized mutual exclusion

## Goal

Removal of the coordinator as bottleneck and single point of failure (not a goal for now: failure tolerance in general)

## Idea

- direct communication between all competitors  
→ `enter` message to everyone
- response to `enter` is
  - `grant`, if there is no conflict
- otherwise conflict handling following a strategy (FCFS)



# Decentralized mutual exclusion

## Requirements

- all competitors know each other ( $\rightarrow$  not anonymous)
- distributed FCFS requires a total order of events (*tpco* or *tco*)
- bookkeeping in each client for each critical section C
  - C.state = {free, occupied, requested}; own state for CS
  - C.time timestamp of `enter`
  - C.rivals set of competitors waiting for `grant`
  - counter for the number of received grants for `enter`

# Decentralized mutual exclusion

## 1. Requesting a critical section

- $C.state \leftarrow \text{requested}$
- $C.time \leftarrow t_{pco}$
- Sending enter message to all other nodes containing the requested CS, the node's ID and the node's time

## 2. Entering a critical section

- wait for  $n-1$  grant messages
- $C.state \leftarrow \text{occupied}$



# Decentralized mutual exclusion

## 3. Leaving a critical section

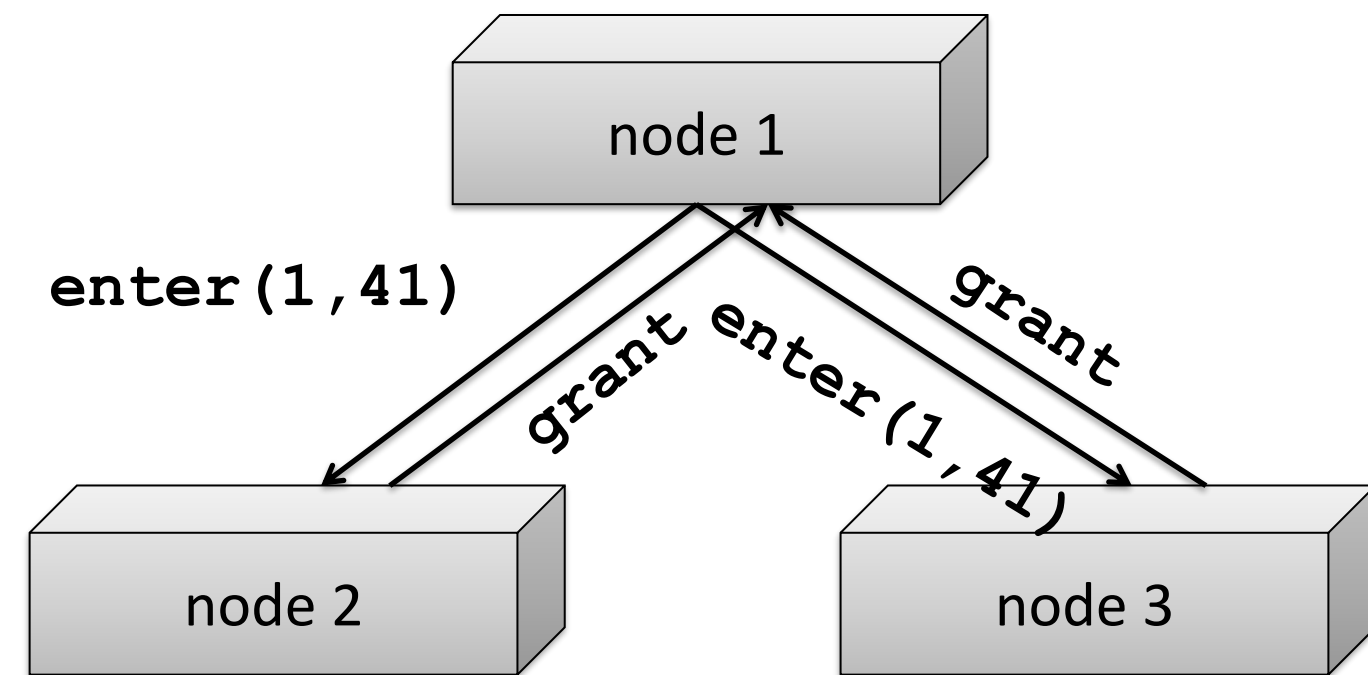
- $C.state \leftarrow \text{free}$
- Sending `grant` message to all nodes in  $C.rivals$

## 4. Response to enter message from node $n_i$ at time $t$

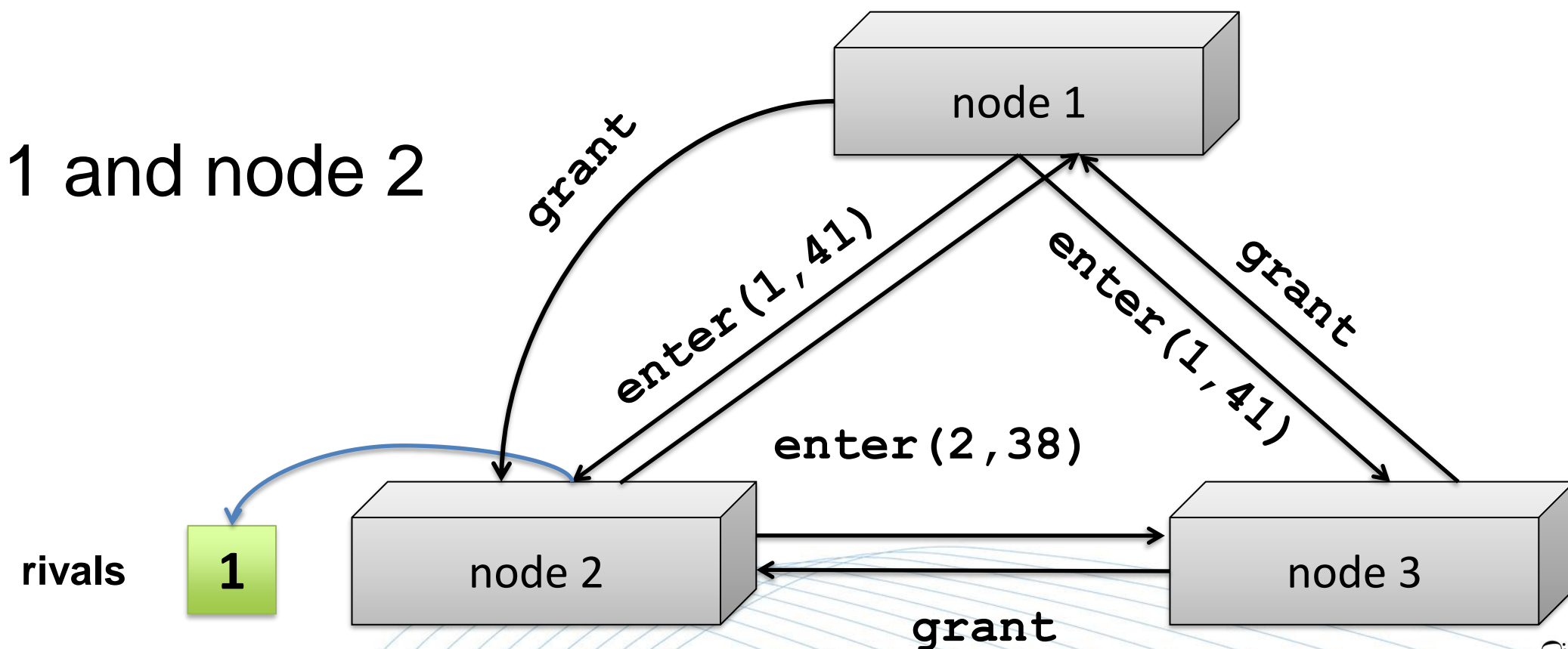
- If  $C.state == \text{free}$   $\rightarrow$  respond with a `grant` message to  $n_i$
- If  $C.state == \text{occupied}$   $\rightarrow$  requesting node is added to  $C.rivals$
- If  $C.state == \text{requested}$  &  $C.time \prec t$   $\rightarrow$  node is added to  $C.rivals$
- If  $C.state == \text{requested}$  &  $t \prec C.time$   $\rightarrow$  `grant` message to  $n_i$

# Example

Node 1 requests the CS at time 41 with no conflict



Conflict between node 1 and node 2





# Comparison to central coordination

- stricter requirements (everyone knows each other, tcpo)
- more complex algorithm
  - hard to prove correctness and liveness
- Fail-stop failures
  - nodes →  $n$  single-point-of-failures instead of 1
  - message loss
    - `enter` → blocks client, but  $n-1$  times more likely
    - `grant` → blocks CS indefinitely

# Comparison to central coordination

## Complexity

- $n$  local states (bookkeeping effort)
- time: 2 operations per CS access  $\rightarrow O(1)$  (but in each node)
- Communication message count per CS access:  
 $2(n-1) \rightarrow O(n)$  instead of  $O(1)$
- Communication message volume per CS access:  
 $O(n)$  instead of  $O(1)$

**Successful disimprovement!**

# Possible Variants

## Implementation

- group communication protocols → reduce communication cost

## System design

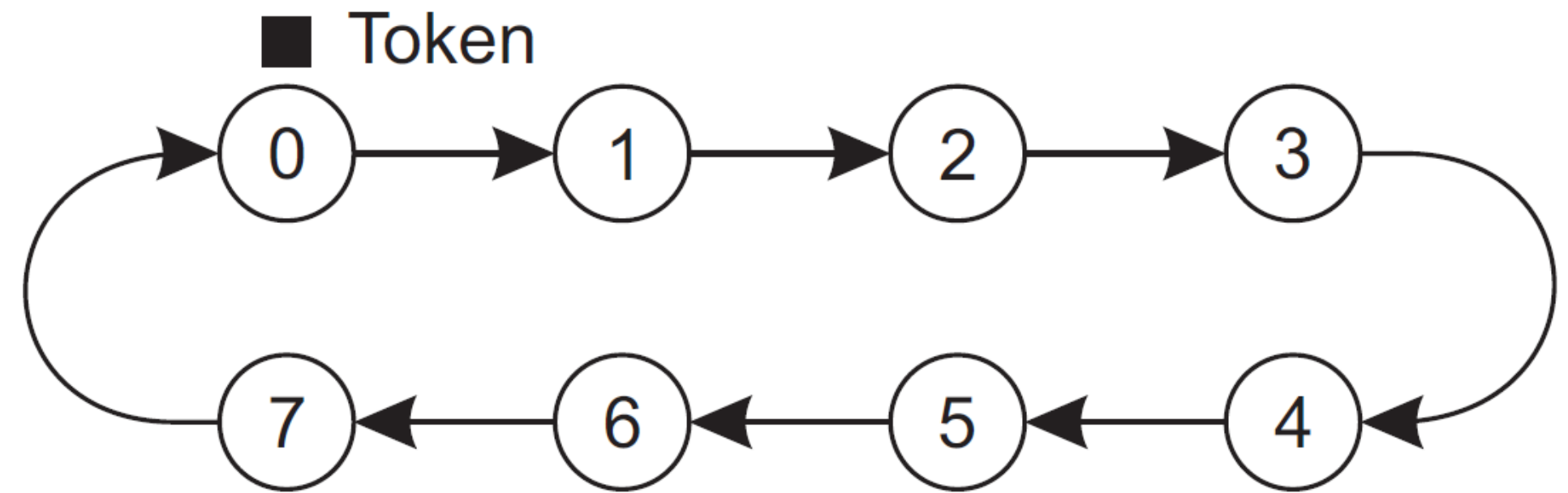
- failure detection and failure handling
- *spof* avoidance (replication or (re-)election of coordinator)

## Algorithm design

- `grant` by not objecting  
(failed nodes and nodes without conflict do not object)
- majority decisions (tolerates a number of failures)
- different communication topology



# Token-ring algorithm



## Idea

- overlay network with ring topology
- for each CS there is a token moving in the ring
- Entering the CS is only possible, if the node is in possession of the token

## Node behaviour

- On receiving a token either enter the CS (only once!) or pass the token along to next node in the ring
- On leaving the CS the token is given to the next node

# Token-ring algorithm

## Pros

- Weaker requirements (anonymous, no order of events required)
- **correctness** and **liveness** are trivial
- **fairness** follows ring topology (not FCFS)
- very efficient in high load scenarios

## Cons

- messages for token passing in low load scenarios
- Fail-stop failures
  - messages → token lost
  - node → ring structure broken, may lose the token

# Election algorithms

## Goal

Instead of using a predefined coordinator the system elects a coordinator and reelects a new one in case of a fail-stop failure of the current coordinator

## Examples

- Mutual exclusion
- Berkeley time synchronization
- super peer selection in P2P networks
- ....



# Election algorithms

## Properties of valid solutions to the coordinator problem

1. At each time, there is never more than one coordinator (**Consensus**)
2. There is always a coordinator after some time (**Liveness**)
3. At some point in time, every operational node knows the coordinator (**Informedness**)

## Common Assumption

- Every node can be coordinator (Homogeneity)

# Bully algorithm

## Idea

The *most important* (i.e. largest index) node alive will get the coordinator role, by gradually telling everyone else who wants to role that they are not getting it. Once everyone else is quiet, the new coordinator ("bully") will inform the other nodes of its victory.

## Requirements

- finite index set with a total order (→ each node has a unique ID)

# Bully algorithm

## Messages and data structure

- three messages (no fail-stop of messages)
  - `elect`
  - `reply`
  - `bully`
- each node  $n_i$  knows the current coordinator  
→ stored in local variable  $c_i$   
(might not be correct at all times)

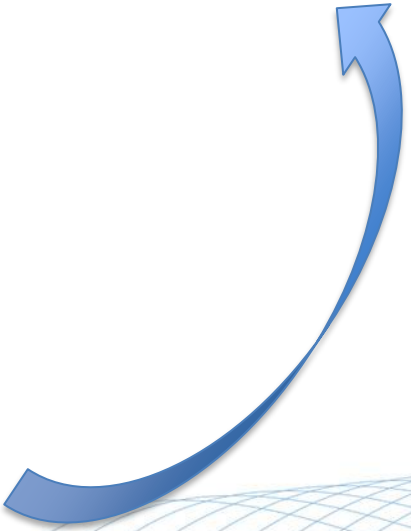


# Bully algorithm

## Starting condition

Either there is no coordinator, one node detects a failure of the current coordinator or a failed coordinator comes back to life

## Algorithm (1)

- Node  $n_i$  initiates election by sending `elect` to all nodes  $n_j$  with  $j > i$  (i.e. with a higher index)
  - Node  $n_j$  receives `elect`
    - `reply` to the sender
    - starts an election on its own
- 

# Bully algorithm

## Algorithm (2)

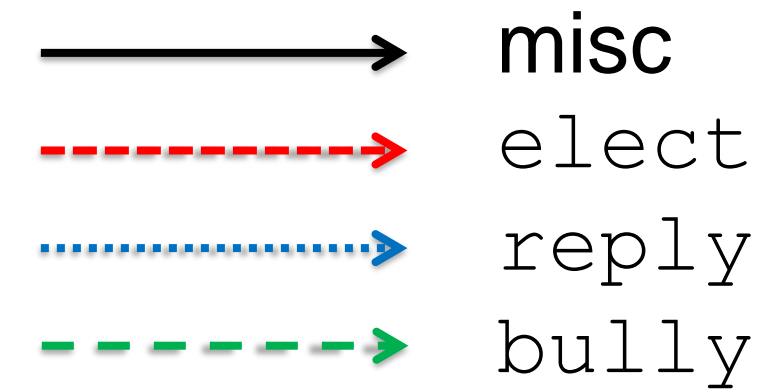
Node  $n_i$

- receives at least one `reply`
  - there is an active node with a higher ID  $\rightarrow$  no further actions
- receives no answer
  - there is no active node with a higher ID  $\rightarrow n_i$  becomes coordinator
  - inform everyone  $\rightarrow \text{bully}(n_i)$  to all nodes

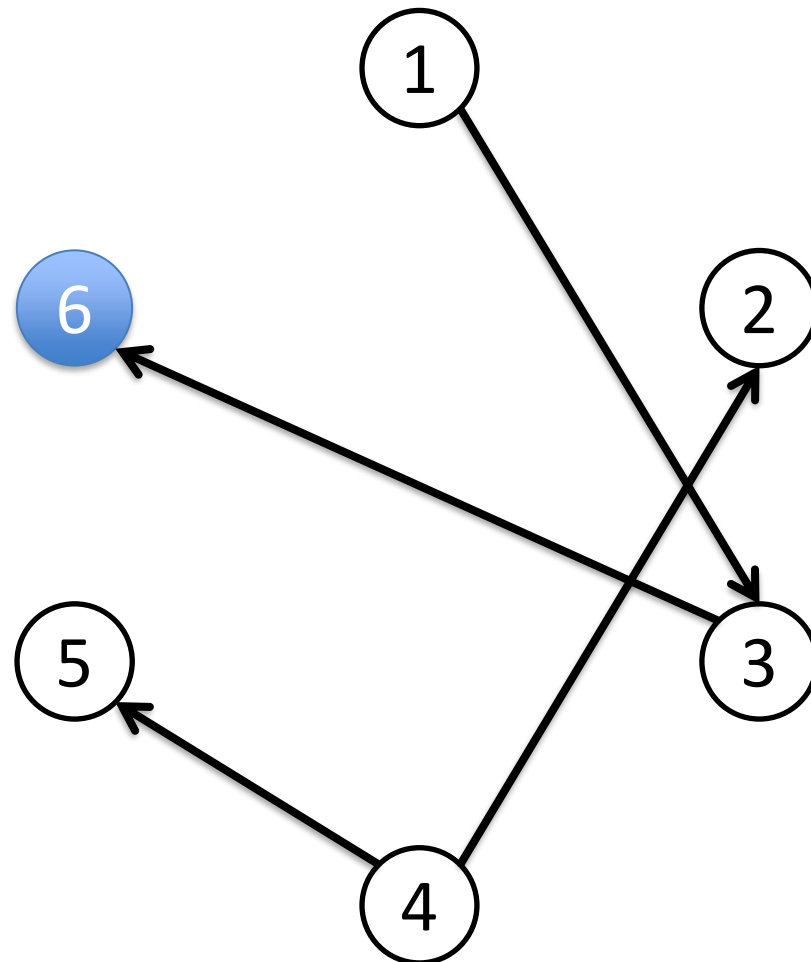
Node  $n_k$

- Receives `bully( $n_i$ )`
  - update  $c_i \leftarrow n_i$

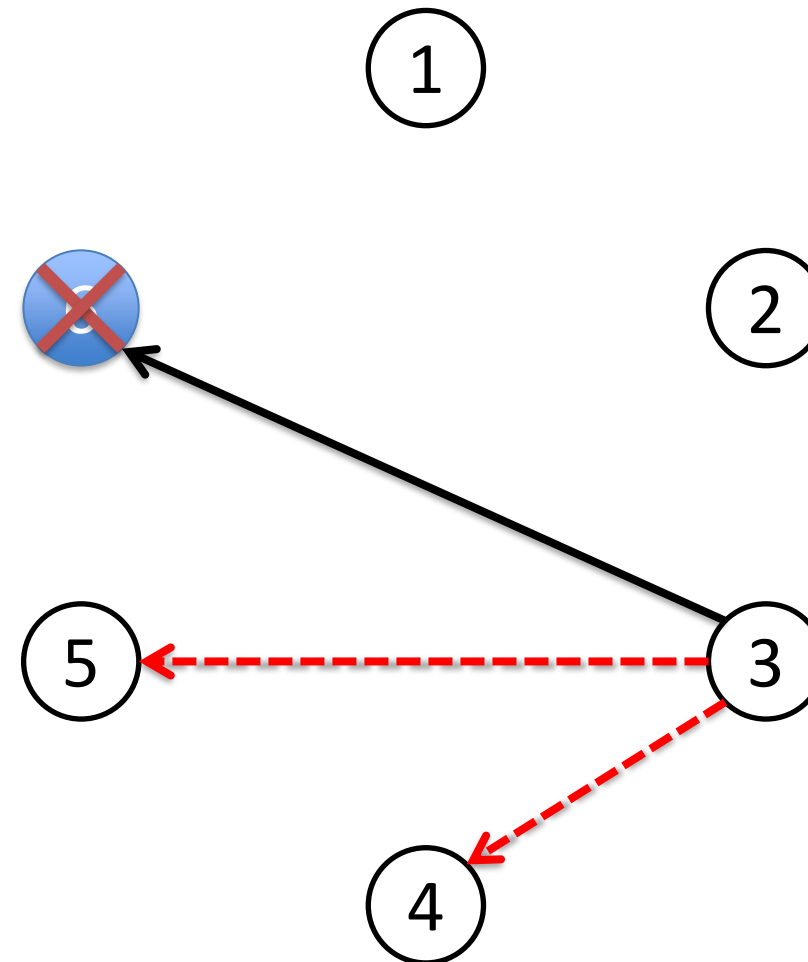
# Bully algorithm - example



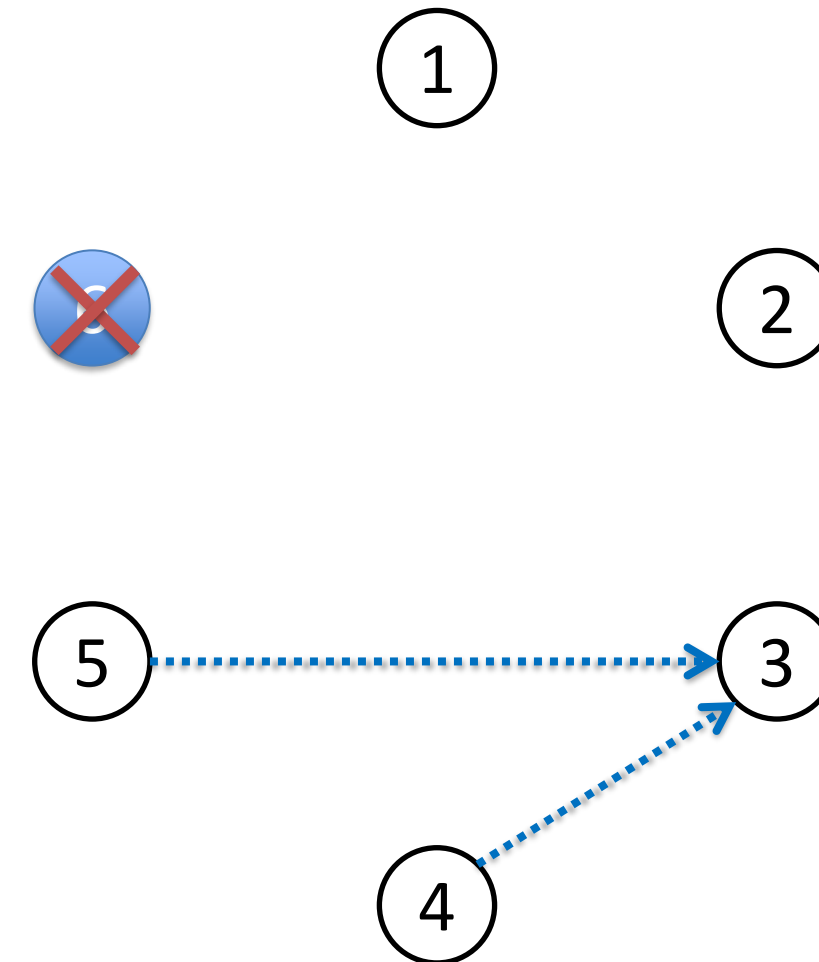
Everything is fine  
6 is coordinator



6 fails, 3 detects that  
and initiates vote

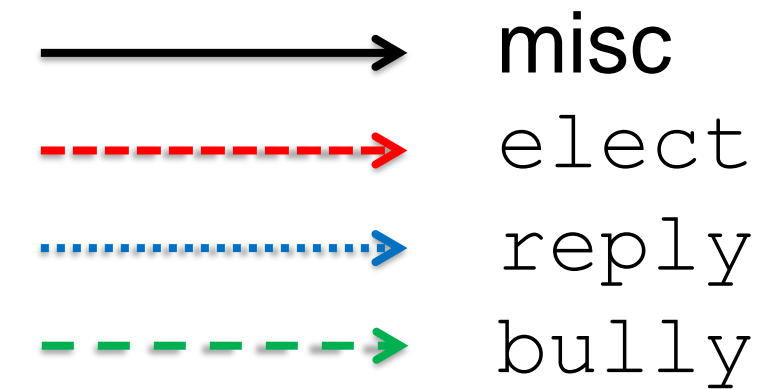


4 and 5 reply  
3 is out of the competition

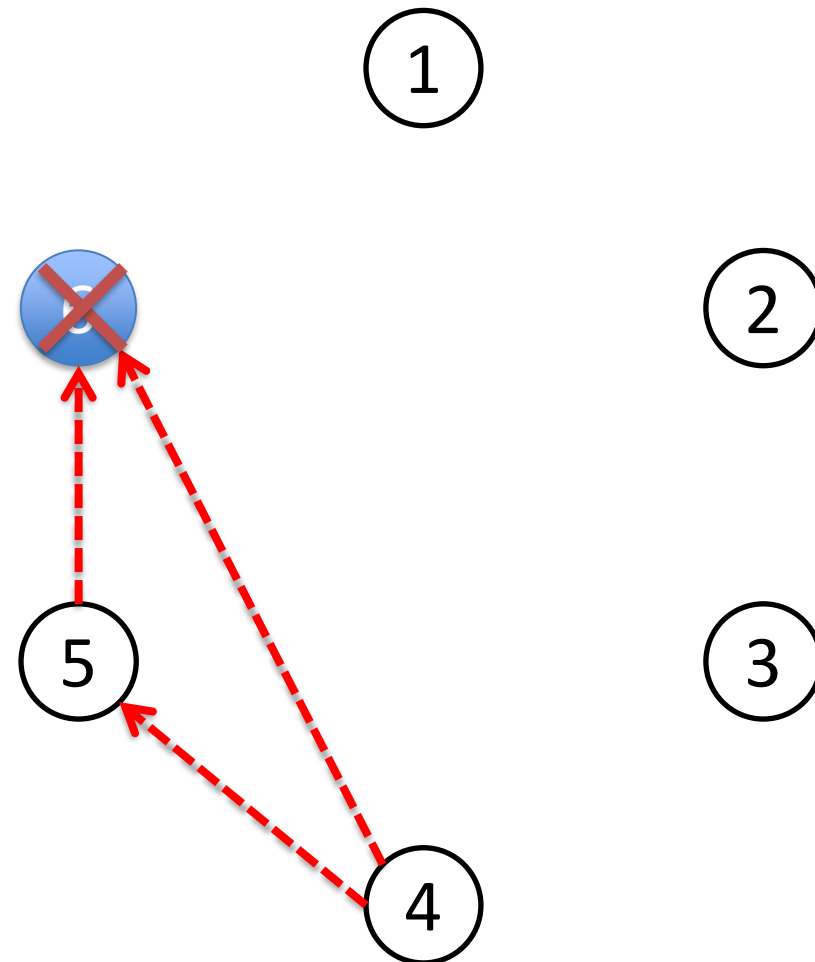




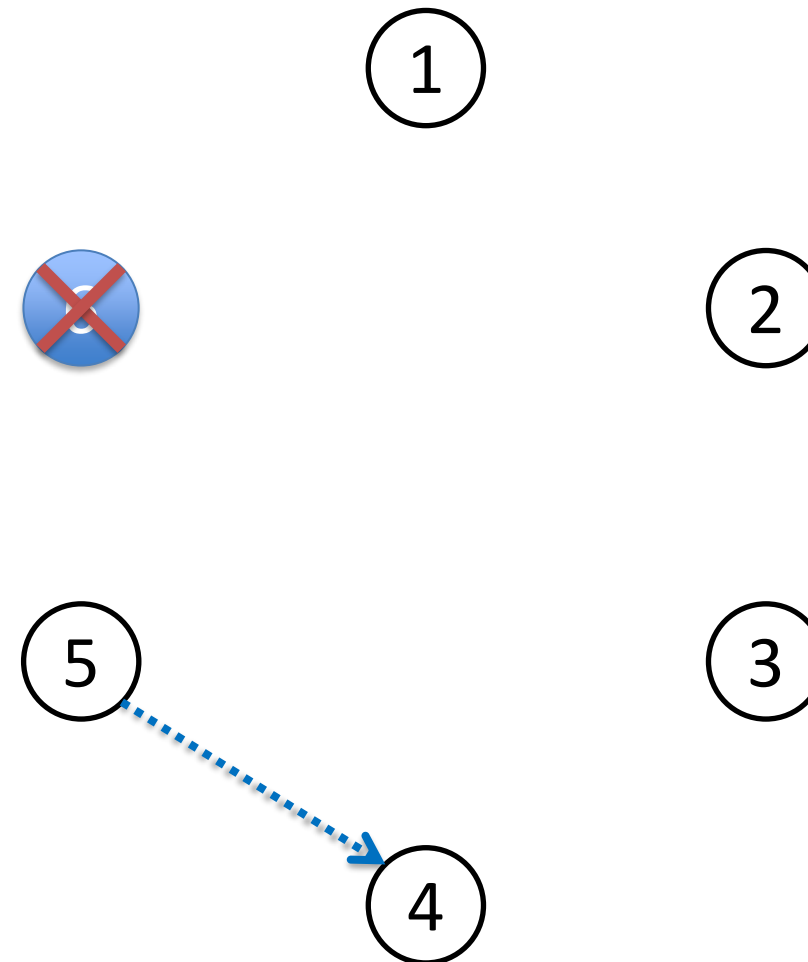
# Bully algorithm - example



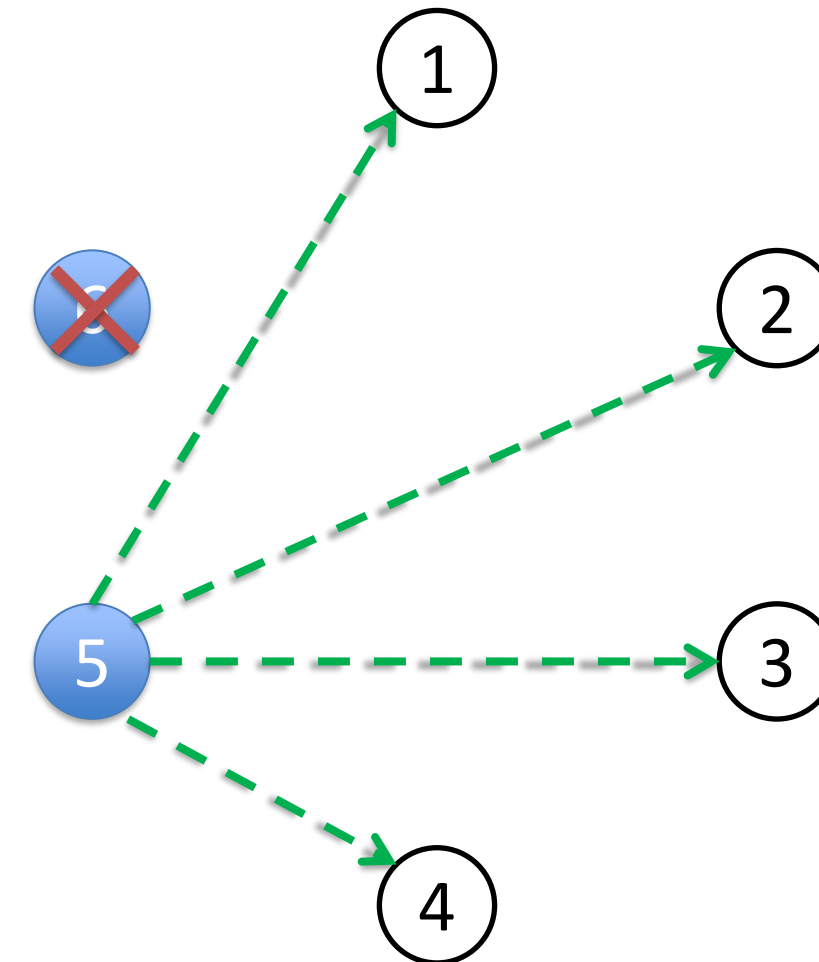
4 and 5 start elections  
of their own



5 replies  
4 is out of the competition



no answer received  
→ bullying



# Bully algorithm – Properties

- **Consensus**
  - multiple, parallel votes – which task does not receive any replies? What are the assumptions here?
- **Liveness**
  - Assumptions for starting the algorithm?
  - Algorithm terminates? And then there is always a coordinator?
- **Informedness**
  - `bully` message

# Bully algorithm – Costs

- **Time complexity**
  - Single failure:  $O(1)$
  - $f$  failures during execution:  $O(f)$
- **Message count & volume**
  - Single failure:  $O(n^2)$
  - $f$  failures during execution:  $O(fn^2)$



# Bully algorithm – Practical Considerations

## Assumptions for detecting of fail-stop coordinator failures

- no message loss
- 'known' response time for a node
- 'known' message trip time

➔ also to determine waiting time for answers to `elect` messages

# Bully algorithm – Practical Considerations

## Idea

Exploit synchronous send operations for failure detection

→ Upper bound for message and response times

- MDT: Maximum Delivery Time
- MHT: Maximum Handling Time

Communication timeout after  $t > 2MDT + MHT$

# Bully algorithm – Practical Considerations

## Estimation of MHT and MDT often difficult

- Maximum Delivery Time
  - WANs without guarantees
  - load in LANs
- Maximum Handling Time
  - server load
  - scheduling strategies (FCFS, priority-based)

➔ Easier in realtime systems



# Bully algorithm – Practical Considerations

## Consequences

- choosing MDT and MHT too large
  - probability for seeming failures low (never zero though)
  - slow to detect actual failures
- choosing MDT and MHT too small
  - fast to detect actual failures
  - probability for seeming failures higher

## Heartbeat ("Are you alive?") messages

- out-of-band data with high priority at the replying side

# Chang's Ring algorithm

## Idea

- overlay network with ring topology
- determine maximum over node IDs
- publication of the maximum ID

## Requirements

- finite index set with a total order
- at least partially known ring topology (successor & successor's successor or a way to determine successors)

# Chang's Ring algorithm

## Messages and data structure

- two messages (no fail-stop of messages)
  - `elect`
  - `meCoordinator`
- each node  $n_i$  knows the current coordinator  
→ stored in local variable  $c_i$   
(might not be correct at all times)



# Chang's Ring algorithm

## Algorithm

- Node  $n_i$  initiates election by sending `elect( $i$ )` to successor node
- Node  $n_j$  receives `elect( $i$ )`
  - if  $i \neq j$ :  $n_j$  sends `elect(max( $i, j$ ))` to successor node
  - if  $i = j$ :  $n_j$  sends `meCoordinator( $i$ )` to successor node
- Node  $n_j$  receives `meCoordinator( $i$ )`
  - if  $i \neq j$ : update  $c_j \leftarrow n_i$  and forward message to successor
  - if  $i = j$ : finished

# Chang's Ring algorithm – Properties

- **Consensus**
  - maximum over the ordered index set is unique
- **Liveness**
  - Assumptions for starting the algorithm?
  - Algorithm terminates? And then there is always a coordinator?
- **Informedness**
  - `meCoordinator` message round
- **Time complexity:**  $O(n)$
- **Message count & volume:**  $O(n)$

# Election algorithms – Summary

## Bully vs. Ring

### Nodes

- finite index set with a total order
- know all nodes vs. know a number (2+) successors

### Failures

- only nodes, no messages loss

### Complexity

- fast, expensive vs. linear growth with size

**➔ No handling of message losses or byzantine failures!**



# Floodset algorithm

## Goal

- Consensus between  $n$  nodes
- tolerant to  $f$  fail-stop failures of nodes and messages (known upper limit of failures,  $f$ -resilience)

## Idea

- building a local repository of all voting information
  - distribution of all the local knowledge
  - in multiple communication rounds
- ➔ maximum redundancy ("flooding")

# Floodset algorithm

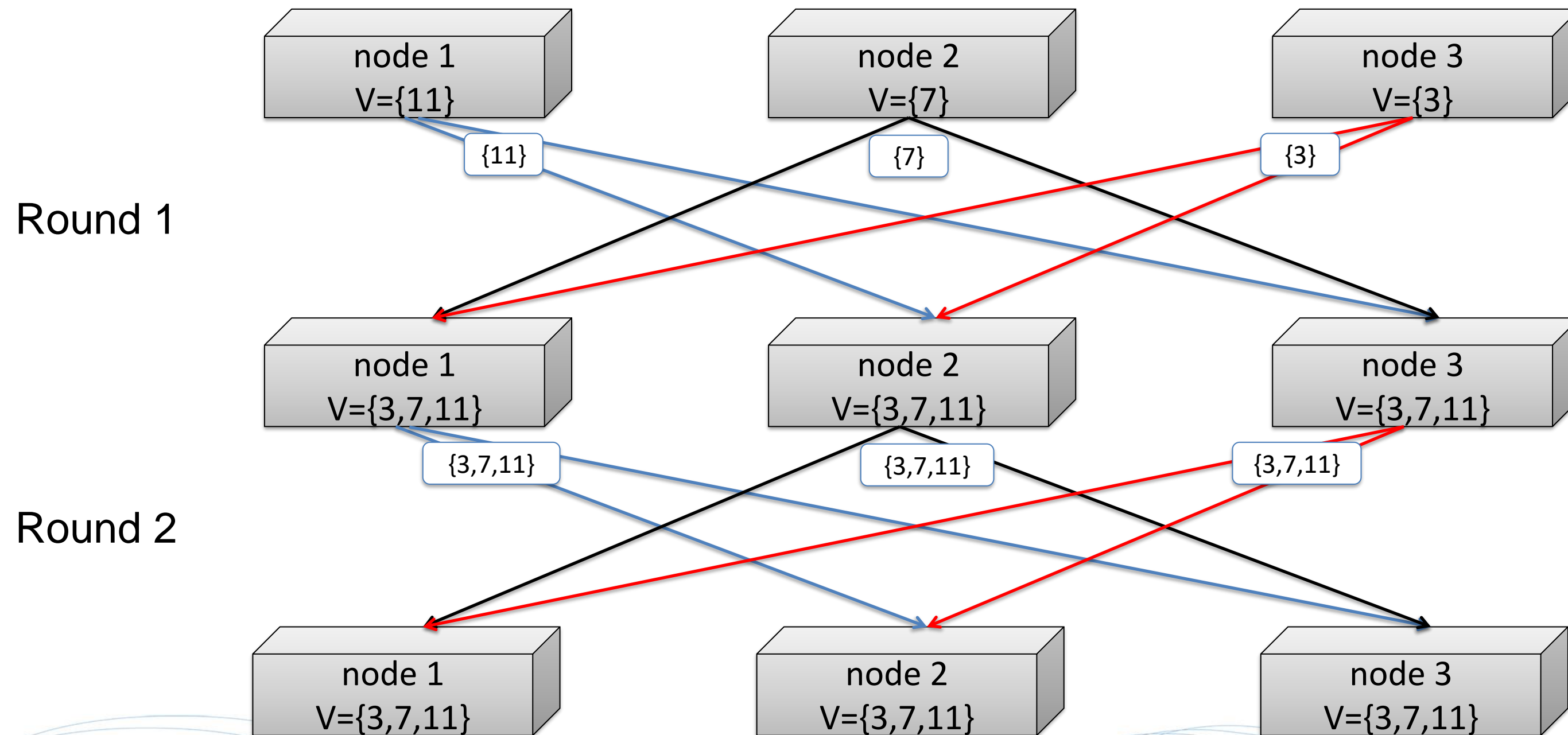
## Algorithm

- **Round 1:** Each node broadcasts their own vote
- **Round 2 to round  $f+1$ :** broadcast all known values
- **End of Round  $f+1$ :** compute consensus result

## Example

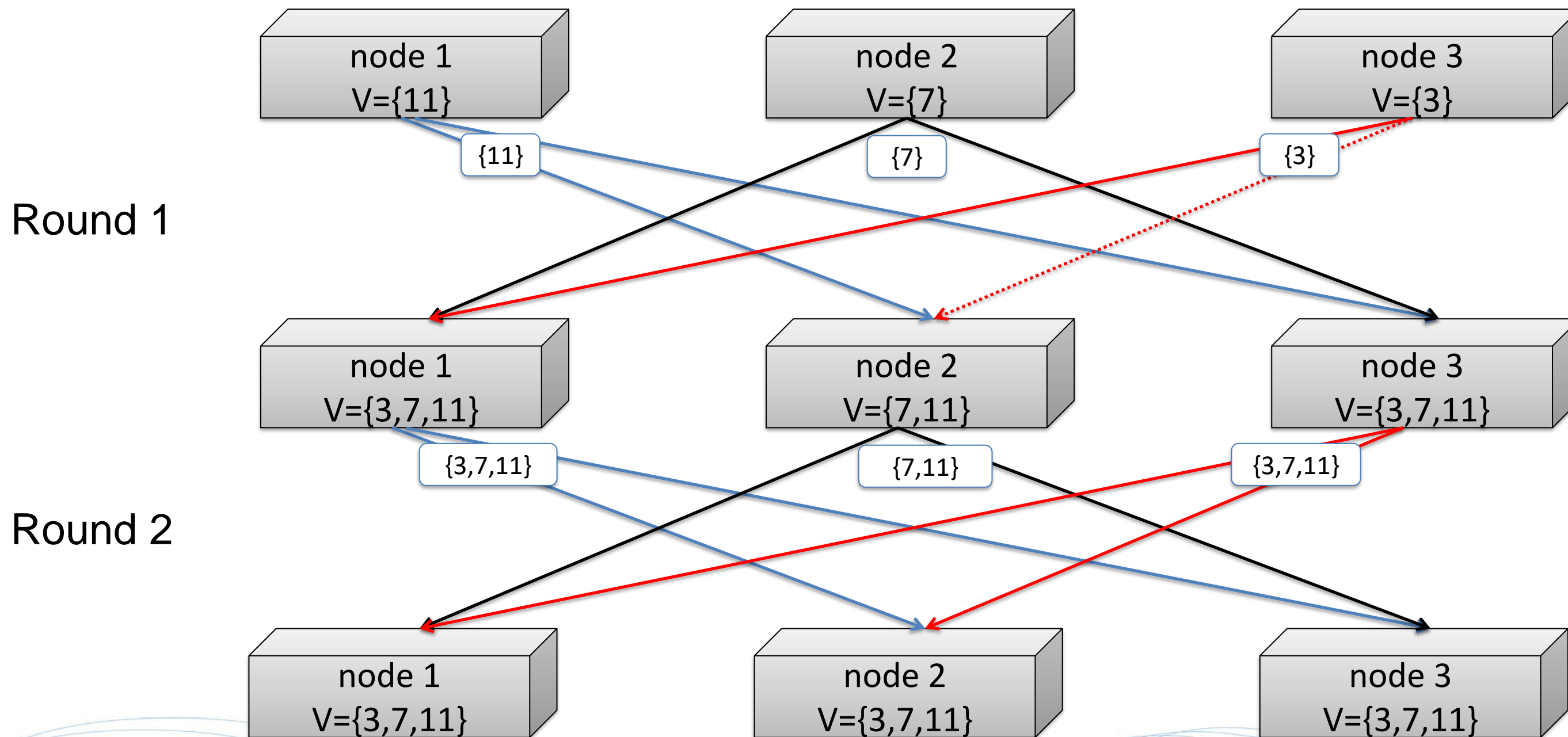
- Agreeing on a minimum state number

# Floodset algorithm - example

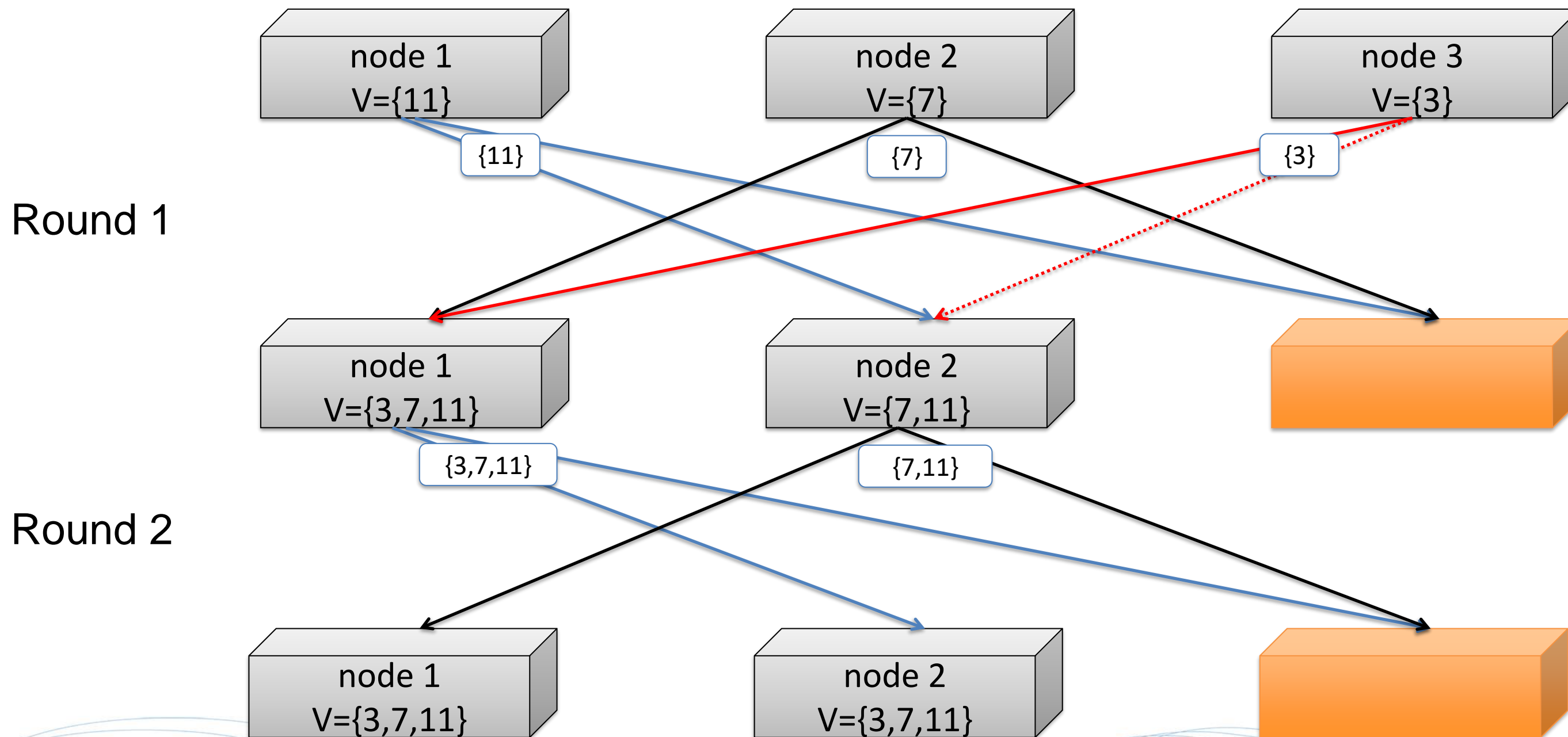




# Floodset algorithm - example



# Floodset algorithm - example



# Observations

- after a failure-free round consensus is achieved
- If the failure-free round occurs before round  $f+1$ , the consensus is maintained for all following rounds
- due to a maximum of  $f$  failures the first guaranteed failure-free round is  $f+1$



# Floodset algorithm

## Complexity

- Time: independent from the number of nodes  $O(f)$
  - Message count  $\leq n(n-1)(f+1) O(fn^2)$
  - Message volume  $O(fn^3)$
- simple & fast
- many messages, assumptions about the upper bound  $f$

**Correctness is provable**

# Exponential Information Gathering

## Requirements

- finite index set with a total order  
( $\rightarrow$  each node has a unique ID)
- each node knows all other nodes
- number of failures has a known upper bound  $f$
- *authentic* messages, but not necessary of *integrity*  
(the sender is identifiable, but manipulations to the messages are not detectable)

# Exponential Information Gathering

## Algorithm

- Each node constructs iteratively a local EIG tree  $T$ 
  - the nodes own vote is the root of the tree
  - round  $k$ ,  $k > 0$ 
    - layer  $k-1$  is sent to all other nodes (flooding)
    - layer  $k$  is constructed with the received messages from other nodes
- with upper limits of failures  $f$ , the number of rounds is  $f+1$
- after  $f+1$  rounds each node has a EIG tree of depth  $f+1$  to determine the voting result



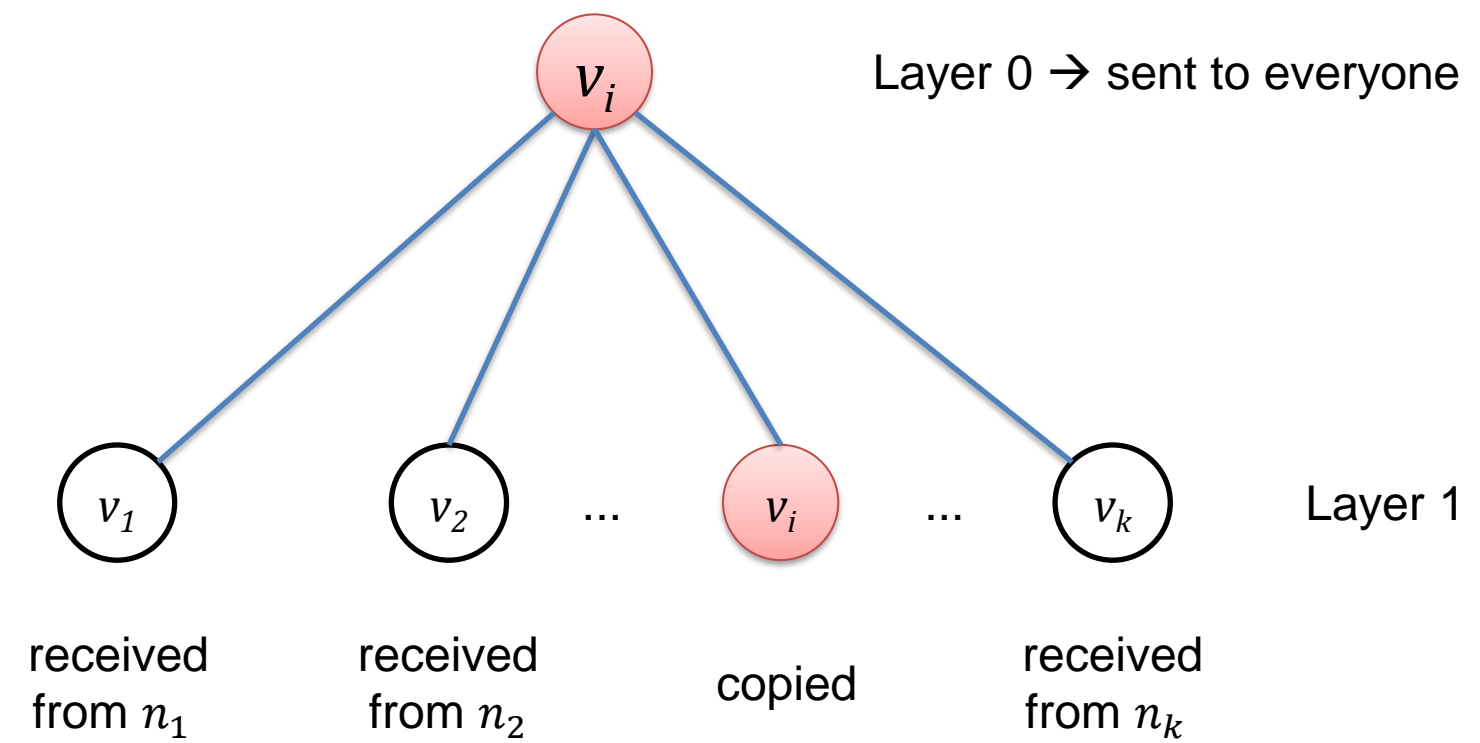
# ELG - example

## Round 1

$k$  nodes in the system

Tree of node  $n_i$

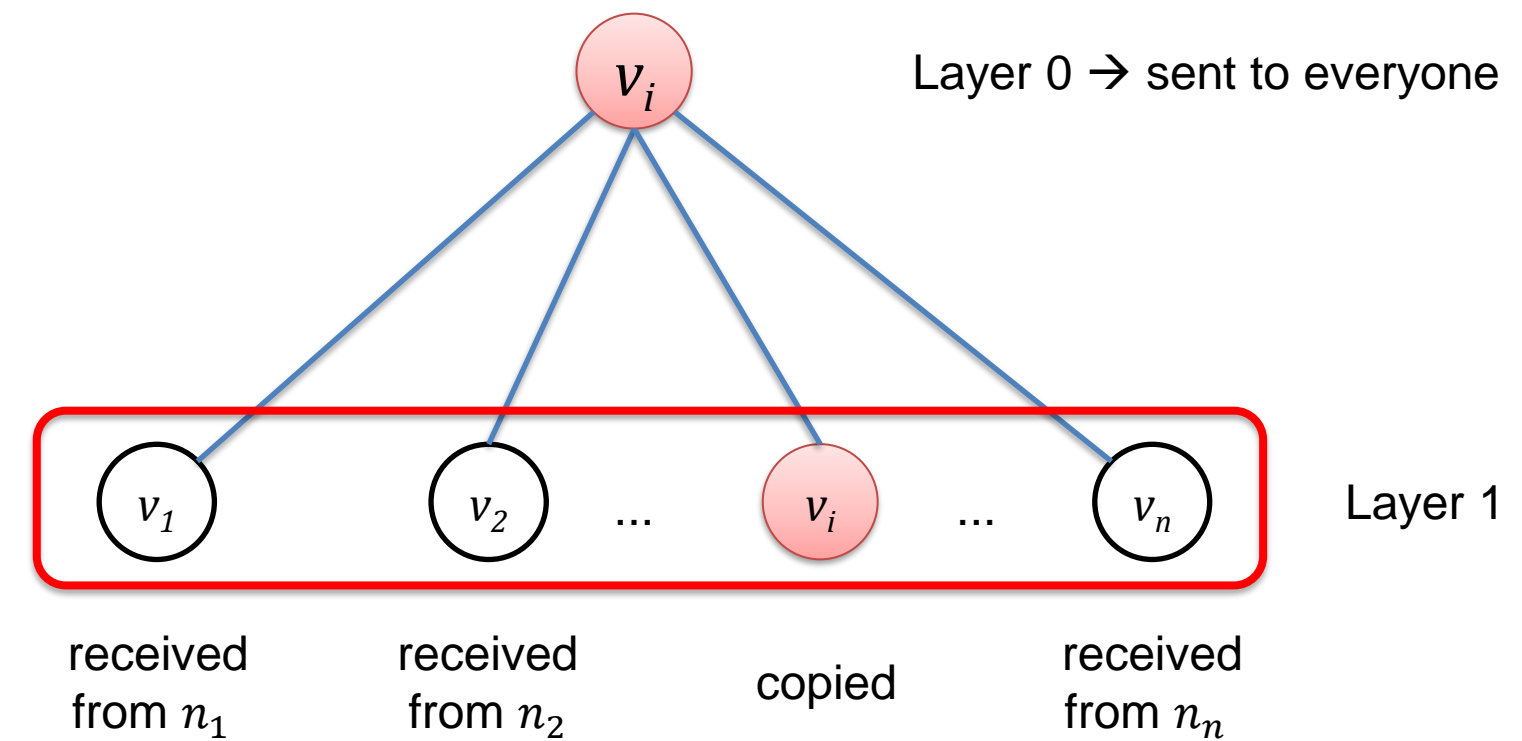
Vote of  $n_i$  denoted by  $v_i$



# ElG - example

## Round 2

- send layer 1 to everyone
- receive
  - from  $n_1$  a tuple  $(n_1: v_1, n_2: v_2, \dots, n_k: v_k)$
  - from  $n_2$  a tuple  $(n_1: v_1, n_2: v_2, \dots, n_k: v_k)$
  - ...
  - from  $n_k$  a tuple  $(n_1: v_1, n_2: v_2, \dots, n_k: v_k)$



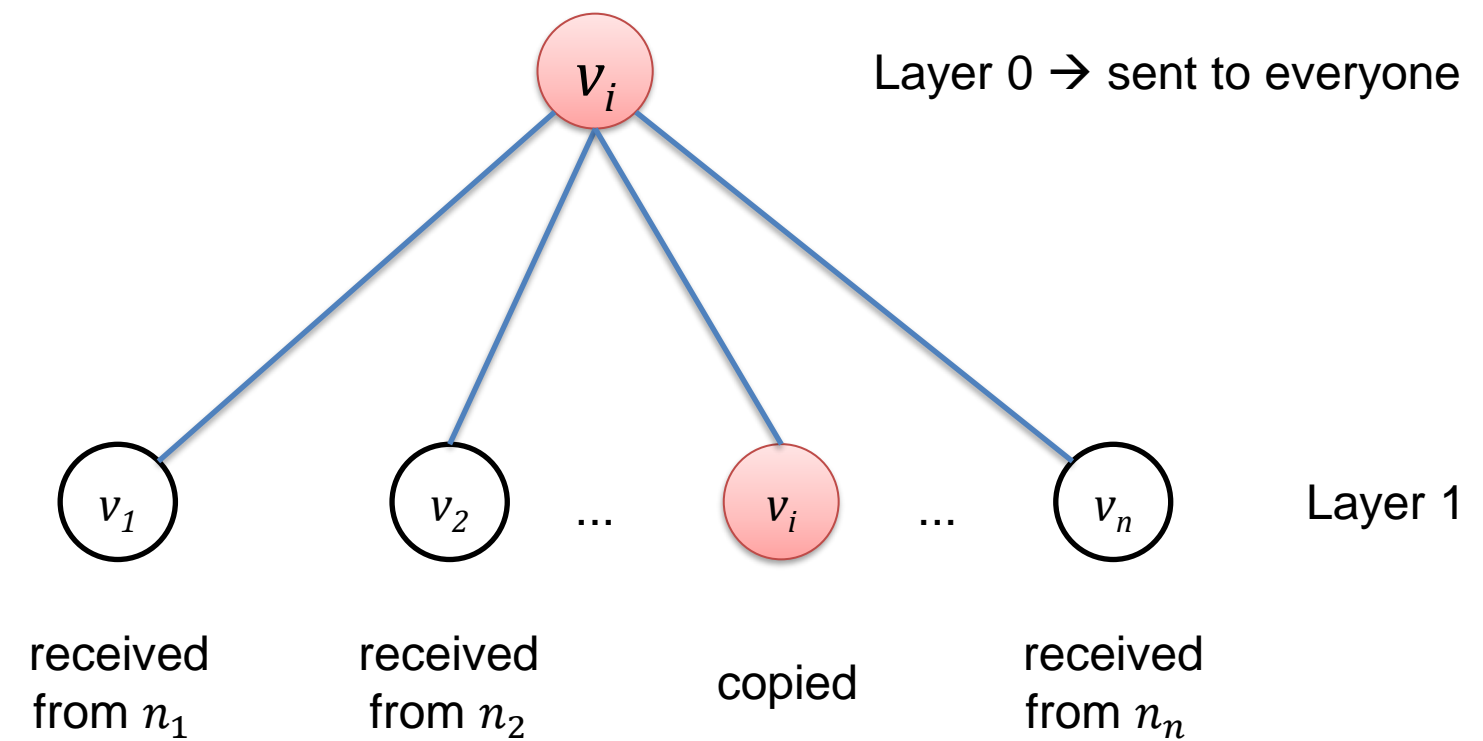
# ElG - example

## Round 2

- construction of layer 2

## Idea

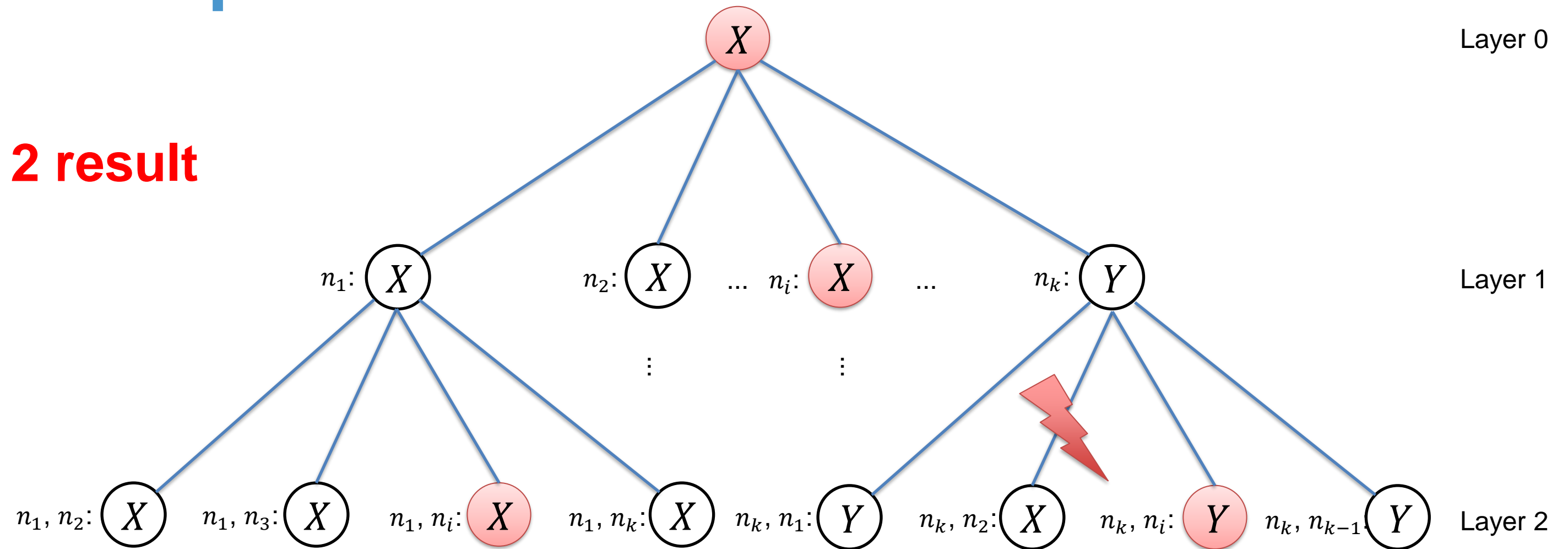
- store *who said what when*
  - *When* encoded in the tree layer
  - *Who* and *what* in the tree node annotation
    - $n_1: v_1 \Leftrightarrow n_1$  says  $v_1$
    - $n_1, n_2, n_3: v_1 \Leftrightarrow n_3$  says that  $n_2$  said that  $n_1$  said  $v_1$





# ElG - example

## Round 2 result



Ups!  $n_k$  told me Y, not X!

- Is  $n_2$  lying to me?
- Did  $n_k$  lie to me?
- Did  $n_k$  lie to  $n_2$ ?

# ElG example observations

## Path to the leafs

- Sequence of communication rounds with statements regarding one vote from different nodes

## Subtrees

- Statements regarding the vote of one node itself (layer 1) and (a sequence of) other nodes

➔ If no one lies or is silent, all votes in a subtree are identical!

# ElG tree construction

## Round 1

- Send vote  $v_i$  to everyone and start tree construction
- create  $n$  tree nodes
- annotation for each node
  - if  $v_j$  arrives from  $n_j$ , tree node  $j$  in  $n_i$  annotates  $(n_j: v_j)$
  - otherwise  $j$  is annotated with  $(n_j: S)$  [S=silent]



# ElG tree construction

## Round $k$ , $2 \leq k \leq f+1$

- Send all annotations  $x:v$  from layer  $k-1$  not containing node  $n_i$  to everyone ( $x$  being a node sequence)
- create  $n^k$  tree nodes on layer  $k$
- annotation for each node
  - if  $x$  arrives from  $n_j$ , tree node  $j$  in subtree annotates  $(n_j:v_j)$
  - otherwise  $j$  is annotated with  $(x: S)$  [ $S$ =silent]

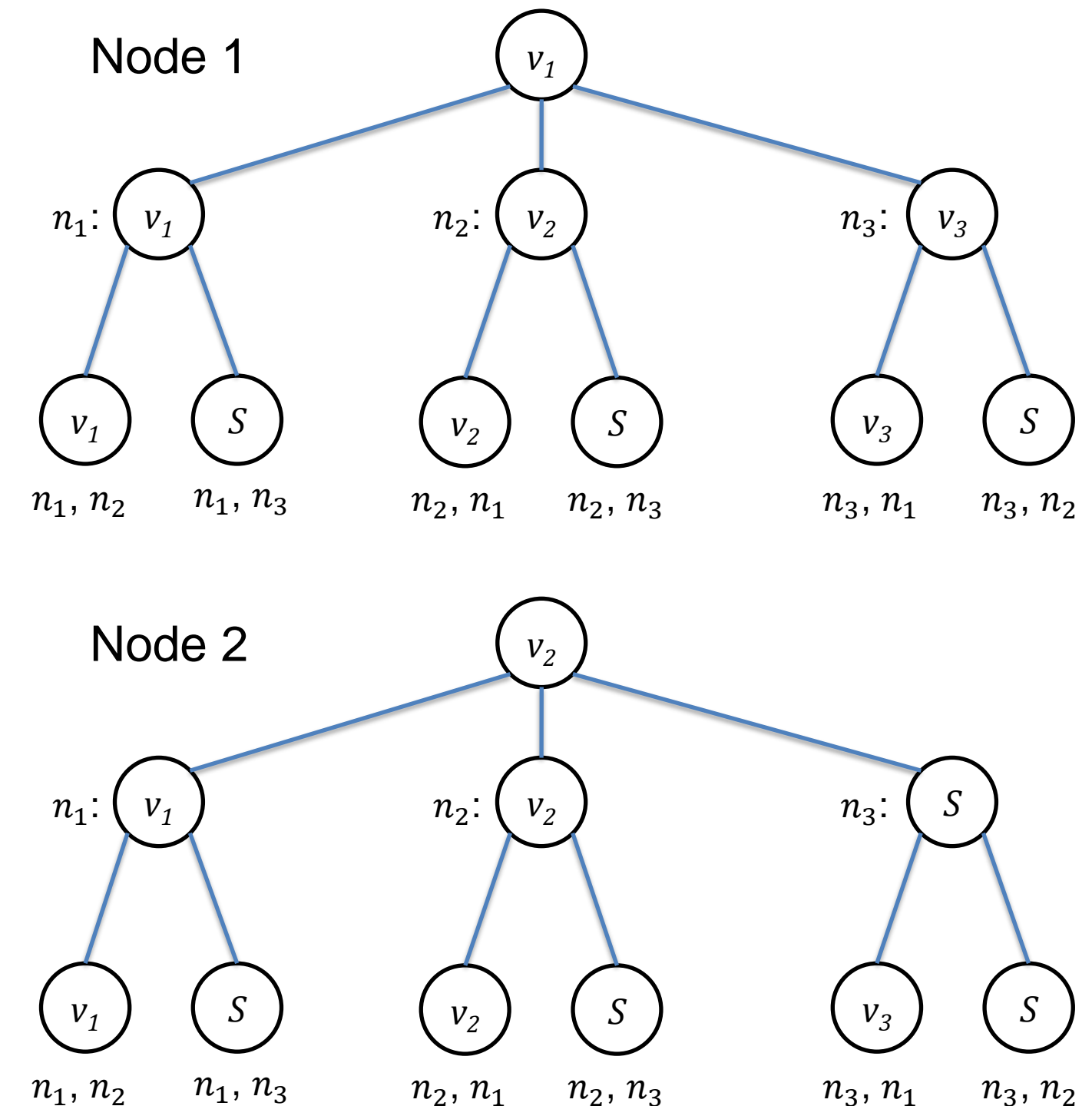
# ELG example

## Scenario

three nodes with node 3 failing in the first round after sending a message to  $n_1$  but before sending it to  $n_2$

See the Floodset example above

→ Same leaf structure in both nodes



# Exponential Information Gathering

## Costs

- **Time:** independent from the number of nodes  $O(f)$
- **Message count**  $\leq n(n-1)(f+1) \rightarrow O(fn^2)$
- **Message volume**  $O(n^f)$  (hence the name)



# Exponential Information Gathering

## Limits

- $n = e + f$ 
  - $n$  total number of tasks
  - $e$  total number of active, non-failed tasks
  - $f$  total number of failures
- $k > 2f$  and  $n > 3f$  ('twice as many active tasks as failures')
- if messages have integrity  
(manipulation during transport is detectable)
- $k > f$  and  $n > 2f$  ('more active tasks than failures')



# The Byzantine Generals Problem

## Scenario

- Assault is only successful if it is conducted at the same time → Consensus
- Communication between the generals using messengers
- $\exists$  corrupt general that sends untrue messages



1st Division



3rd Division



2nd Division



4th Division



# The Byzantine Generals Problem

Toy problem representing the worst kind of failures

- Nodes actively and intelligently sabotaging consensus efforts
- Messages might be intercepted or changed

What are the requirements, limits, costs and ways to achieve consensus among non-failed nodes?



# Consensus with byzantine failures

A distributed algorithm achieves consensus in the presence of byzantine failures iff

1. No two not-failed nodes reach a different result  
(**Consensus**)
2. All not-failed nodes reach at some time a result  
(**Termination**)
3. If all tasks vote 'x', then 'x' is the only valid result  
(**Permissibility**)

# Consensus with byzantine failures

## Idea

- building a local repository of all voting information
- distribution of all the local knowledge
- in multiple communication rounds

→ maximum redundancy

→ expensive

# A consensus algorithm

## Three steps

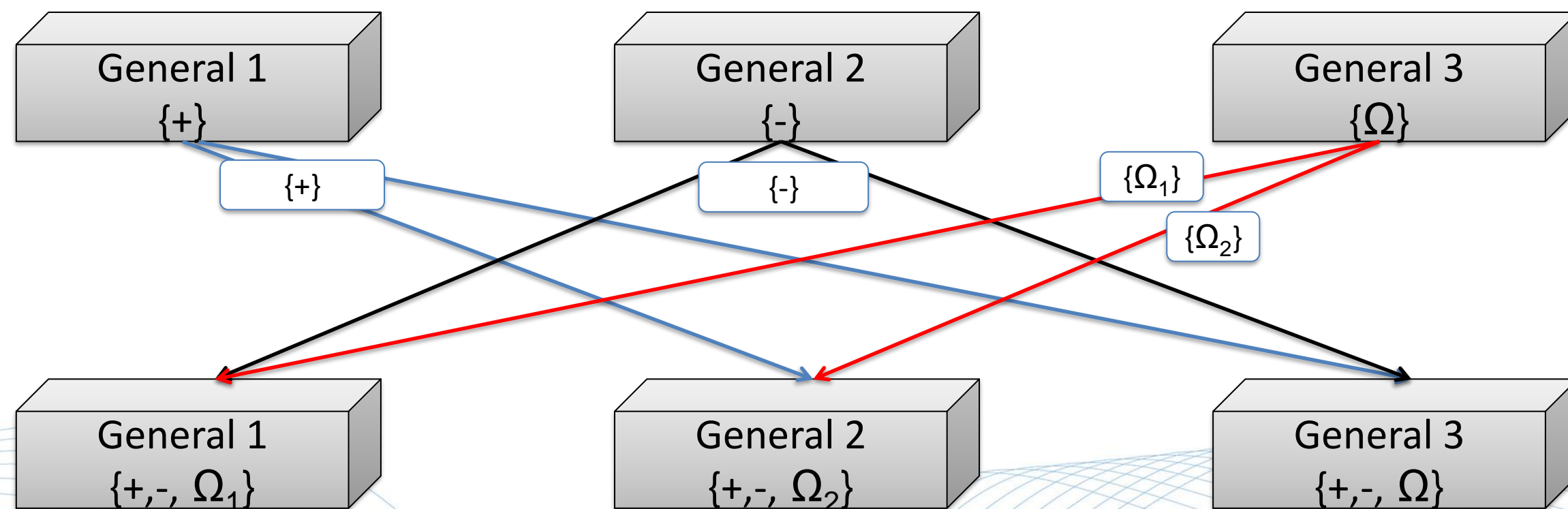
1. Exchange of opinions
  - every general sends their vote to everyone else
2. Validation
  - all generals send the information received from step 1 to everyone else
3. Decision
  - each general decides based on the received information



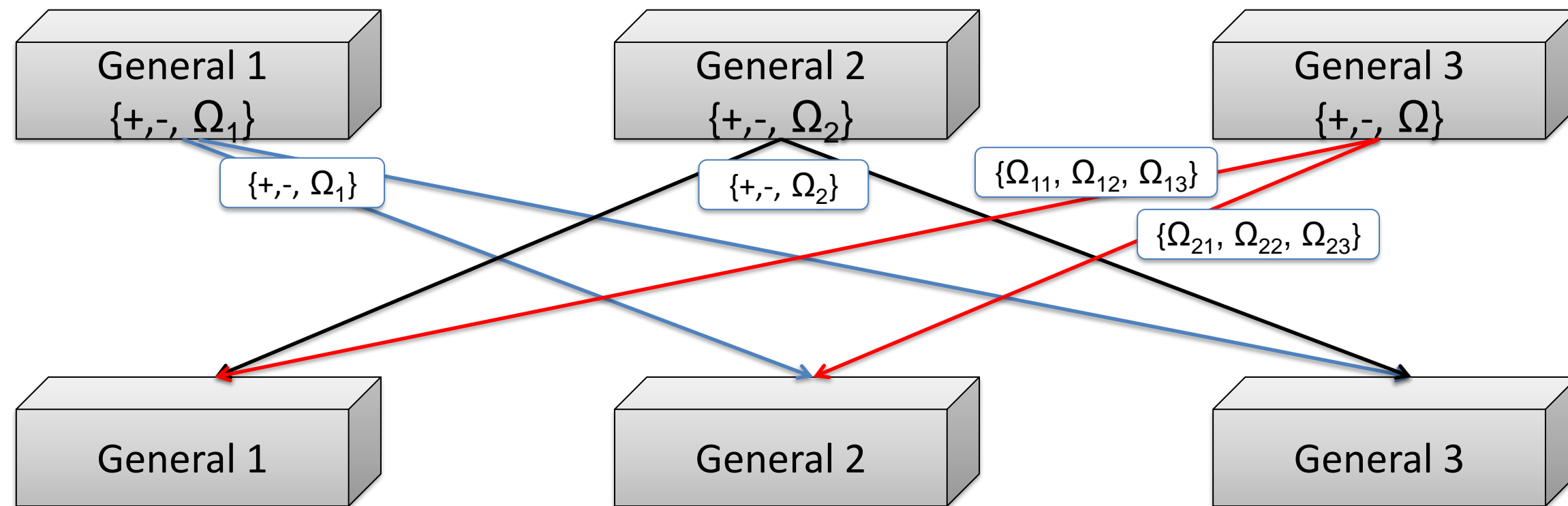
# 1. Exchange of Opinions

## Three generals

- General 1 (loyal): + (for attacking)
- General 2 (loyal): - (against attacking)
- General 3 (traitor): messages to frustrate consensus



## 2. Validation



### Knowledge of General 1

- $\{+, -, \Omega_1\}$
- $\{+, -, \Omega_2\}$
- $\{\Omega_{11}, \Omega_{12}, \Omega_{13}\}$

### Knowledge of General 2

- $\{+, -, \Omega_2\}$
- $\{+, -, \Omega_1\}$
- $\{\Omega_{21}, \Omega_{22}, \Omega_{23}\}$

### 3. Decision

#### Calculating the decision vector

Simple majority over columns

##### General 1

- $\{+, -, \Omega_1\}$
  - $\{+, -, \Omega_2\}$
  - $\{\Omega_{11}, \Omega_{12}, \Omega_{13}\}$
- 
- $\Sigma \quad \{+, -, x_1\}$

##### General 2

- $\{+, -, \Omega_2\}$
  - $\{+, -, \Omega_1\}$
  - $\{\Omega_{21}, \Omega_{22}, \Omega_{23}\}$
- 
- $\Sigma \quad \{+, -, x_2\}$

**Case 1:** traitor did not lie in round 1:  $\Omega_1 = \Omega_2$

$\Rightarrow x_1 = x_2 \Rightarrow$  same decision for General 1 and 2



### 3. Decision

#### General 1

- $\{+, -, \Omega_1\}$
  - $\{+, -, \Omega_2\}$
  - $\{\Omega_{11}, \Omega_{12}, \Omega_{13}\}$
- 
- $\Sigma \quad \{+, -, x_1\}$

#### General 2

- $\{+, -, \Omega_2\}$
  - $\{+, -, \Omega_1\}$
  - $\{\Omega_{21}, \Omega_{22}, \Omega_{23}\}$
- 
- $\Sigma \quad \{+, -, x_2\}$

**Case 2:** traitor did ONLY lie in round 1:  $\Omega_1 \neq \Omega_2$

Discrepancy is visible in each node  $\rightarrow$  traitor is identifiable

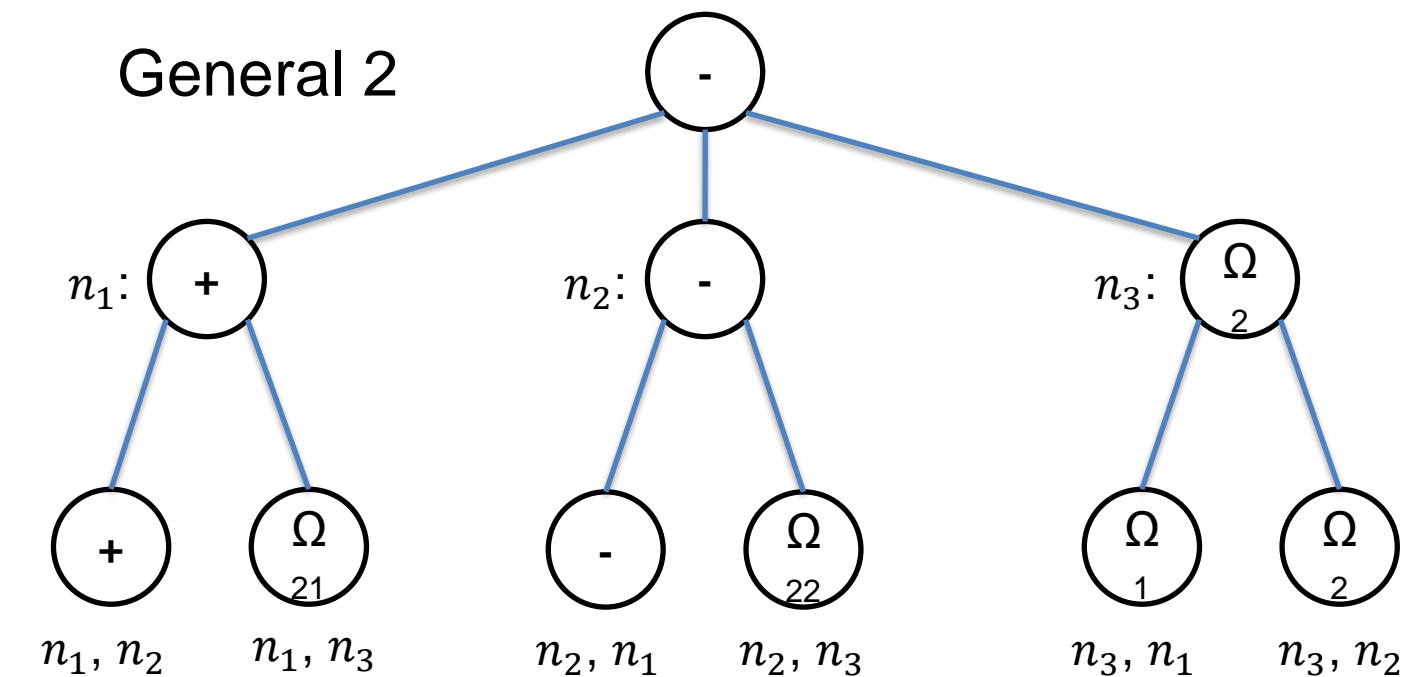
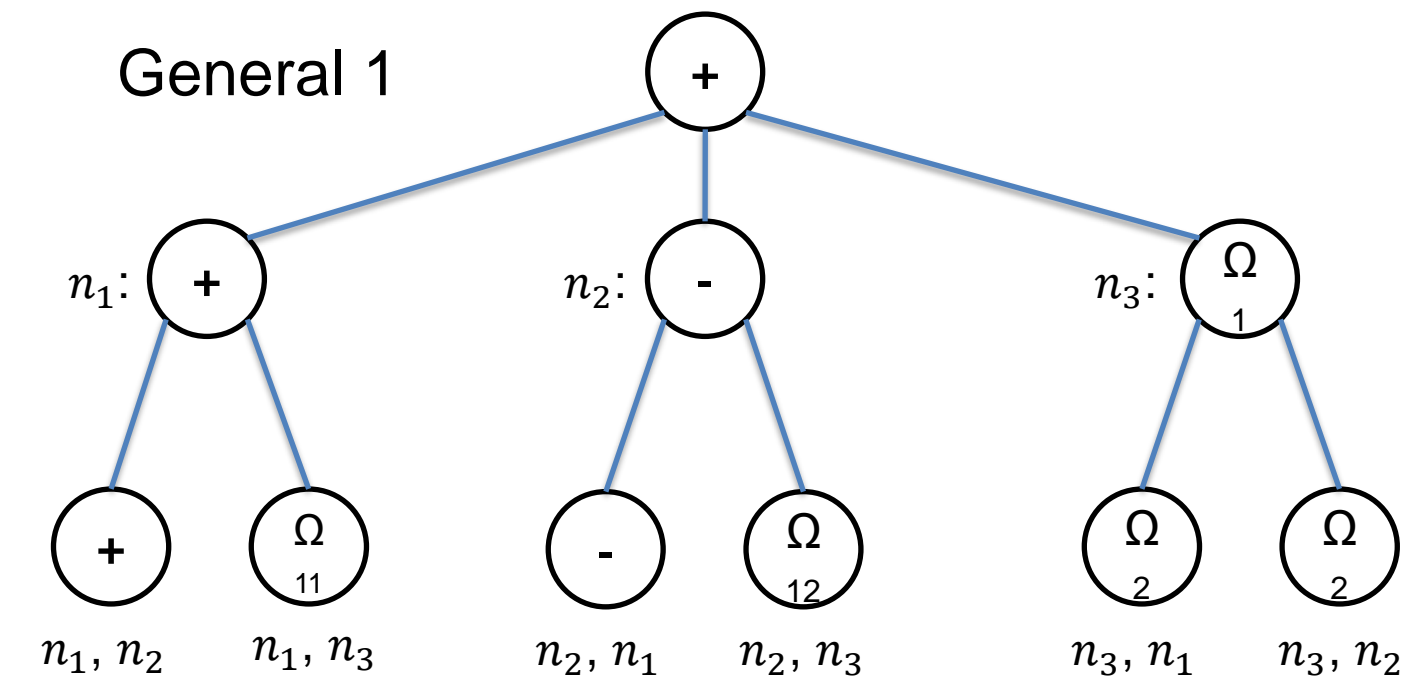
## ELG view

### Detecting lies

If General 3 lies about  $\Omega_{11}$  or  $\Omega_{22}$   
→ directly detectable

If General 3 lies about  $\Omega_{12}$  or  $\Omega_{21}$   
→ not detectable, but doesn't influence majority vote

If General 3 lies about  $\Omega_1$  or  $\Omega_2$   
→ detectable in the  $n_3$  subtree



Node 3



# ElG properties

1. No two not-failed nodes reach a different result (**Consensus**) → same as Floodset
2. All not-failed nodes reach at some time a result (**Termination**) → 2 rounds
3. If all tasks vote 'x', then 'x' is the only valid result (**Permissibility**) → true for simple majority

## Practical consideration

authentic messages of integrity → digital signatures



# Summary on Achieving Consensus

**Without failures:** e.g. flooding in one round

**Fail-stop of nodes:** Bully, Ring, Floodset, EIG

**Fail-stop of messages:** impossible without restrictions

- Upper limit assumption → FloodSet, EIG
- Probabilistic algorithms that have a non-zero probability to not achieve consensus

**Byzantine failures**

- EIG

**Costs and limits**