Distributed Systems

universitätbonn

Chapter 2 – Basic Functionality

Chapter 3 – Coordination

- Time and Global States
- Process Synchronization
- Distributed Transactions

Chapter 4 – Quality of Service

Chapter 5 – Middleware

3.2 Process Synchronization

- Mutual Exclusion and Voting
- Election of Coordinators
- Consensus Problems

Mutual Exclusion

Resources accessed by multiple processes have to be protected against simultaneous accesses

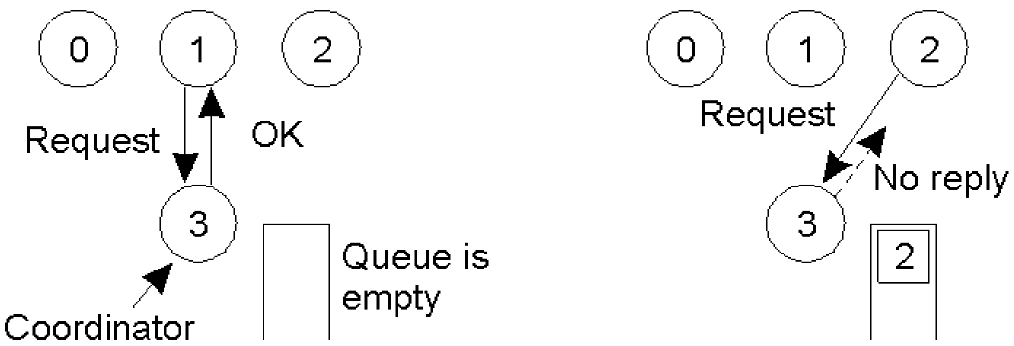
⇨ *Critical regions*

- In one machine: semaphores, mutexes, monitors, ...
- Other concepts are needed in distributed systems
- *Assumption*: message delivery is reliable and processes will not fail, for simplicity there is only one critical region
- Conditions for evaluation:
  - Fairness
  - Starvation
  - Deadlocks
  - Robustness
  - Performance: bandwidth, delay, ...

A Centralised Algorithm

Straightforward: *simulate a single system*

- One process becomes a *coordinator* which controls access operations and keeps a queue for processes which want to enter the critical region
- A process which wants to enter a critical region, asks the coordinator for permission
- If there is no more process in the critical region, the requestor gets an OK from the coordinator
- Otherwise, the request is queued. When the critical region is left by the holding process, the coordinator takes the oldest request from its queue and sends back an OK to the sender of the request.



A Centralised Algorithm

Advantages:

+ Guaranteed exclusive access by centralised control

+ Fair algorithm guaranteeing order of requests

+ No starvation of single processes

+ Easy to implement

+ Only three messages per entry in the critical region

Disadvantages:

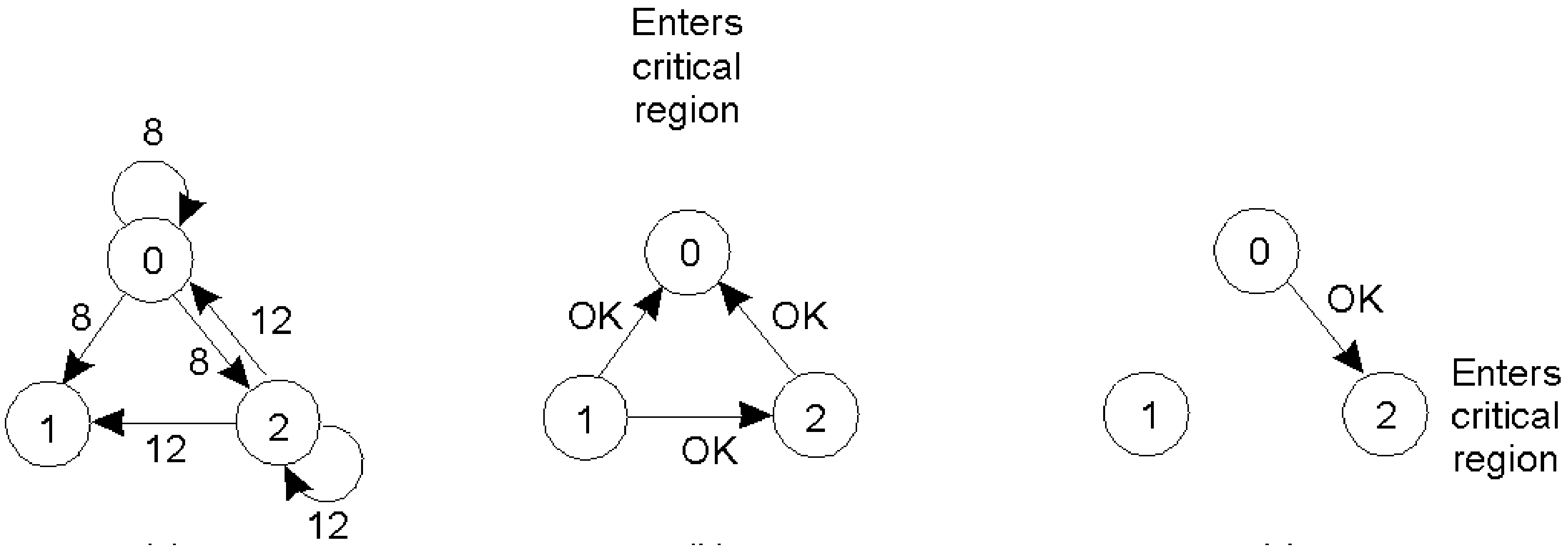
- Coordinator becomes a single point of failure and a performance bottleneck
- It is hard to see, if the coordinator is blocked or crashed (in this case, a new coordinator has to be determined)

A Distributed Algorithm

Ricart/Agrawala: assume that there is a total ordering of all events in a system (e.g. using Lamport timestamps)

- Process *P* wants to enter a critical region; it sends a message containing its process number and timestamp to all processes
- A process *Q* receiving the message of *P* distinguishes
  - If *Q* is not in the critical region and wants not to enter, it answers OK
  - If *Q* is in the critical region, it queues the request until it leaves the region
  - If *Q* wants to enter the critical region, it compares *P*'s timestamp with its own (sent in an own message to all processes). The lower timestamp wins, *Q* answers with OK resp. queues the request
- A process receiving OK from all processes can enter the critical region
- When leaving the critical region, an OK is sent to all processes with requests in its queue

Example for Distributed Algorithm



Characteristics of the algorithm

- No deadlocks, no starvation, but 2(*n*-1) messages for *n* processes
- And: multiple-point-of-failure: not responding means denying permission...
- And: if no group communication is available, each process must manage groups...
- And: overloaded processes are performance bottlenecks for the whole mechanism...
- At the end: slower, more complex, less robust than the centralised algorithm...

Ricart and Agrawala's Algorithm

On initialization
state := RELEASED;

To enter the section
state := WANTED;
Multicast *request* to all processes; request processing deferred here
T := request's timestamp;
Wait until (number of replies received = (*N* – 1));
state := HELD;

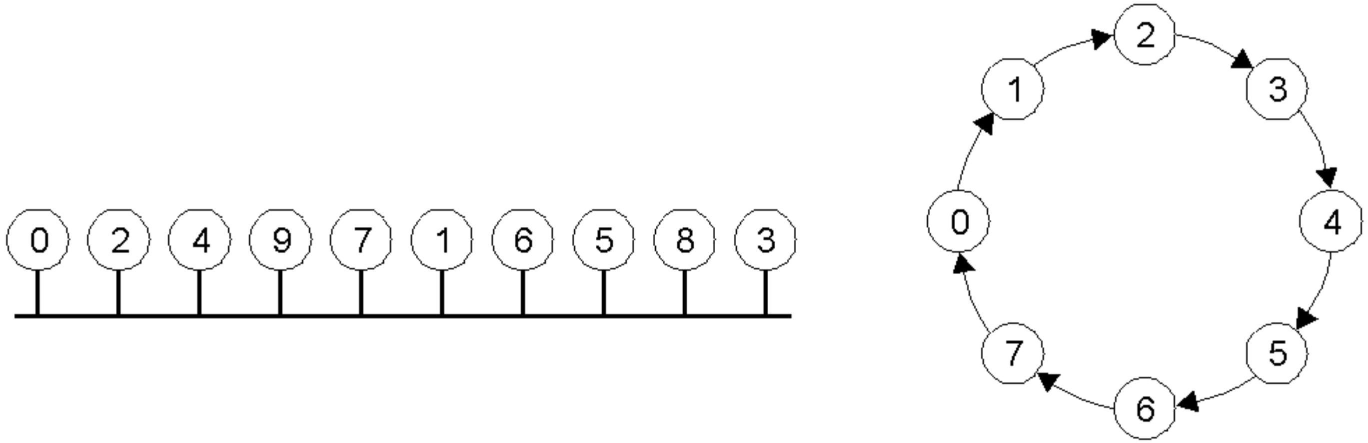
On receipt of a request $<T_i, p_i>$ *at* p_j $(i \neq j)$
if (*state* = HELD or (*state* = WANTED *and* $(T, p_j) < (T_i, p_i)$))
then
queue *request* from p_i without replying;
else
reply immediately to p_i;
end if

To exit the critical section
state := RELEASED;
reply to any queued requests;

A Token Ring Algorithm

Completely different algorithm:

- Construct a logical ring from all processes, with each process knows the successor
- On initializing, process 0 gets a token, which circulates around the ring
- Having the token, a process is allowed once to enter a critical region. After leaving it, the token is passed on



- Advantages: no synchronization delay, no starvation, no deadlocks
- Problems: no real-world ordering of entry requests, token loss, token duplication, process crashes, maintenance of the current ring configuration

Comparison

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized	3	2	Coordinator crash
Distributed	$2 (n – 1)$	$2 (n – 1)$	Crash of any process
Token ring	1 to ∞	0 to $n – 1$	Lost token, process crash

The centralised algorithm seems to be best...

Maekawa's Voting Algorithm

It is not necessary to have a permission from all processes before entering a critical region; permissions are only needed from subsets (which have to have overlaps)

Associate a voting set V_i with each process p_i, where $V_i \subseteq \{p_1, p_2, ..., p_N\}$ so that for all $i,j = 1, 2, ..., N$

- $p_i \in V_i$
- $V_i \cap V_j \neq \varnothing$ (no disjunctive sets)
- $|V_i| = K$ (each set has same size)
- each p_j is contained in M of the voting sets

(optimal solution with minimal K: $K \sim N^{-1/2}$, $M = K$)

[non-trivial problem: how to calculate the optimal sets?]

It works, because by $V_i \cap V_j \neq \varnothing$ there is one process in the intersection only voting for a process in *one* of the subsets.

... but: deadlocks are possible! (but adaptation of the algorithm is possible)

Only $3N^{-1/2}$ messages, which is smaller than $2(N-1)$ for $N > 4$

Maekawa's Algorithm – Part 1

```
On initialization
    state := RELEASED;
    voted := FALSE;

For p_i to enter the critical section
    state := WANTED;
    Multicast request to all processes in V_i;
    Wait until (number of replies received = K);
    state := HELD;

On receipt of a request from p_i at p_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
```

All processes in V_i have answered

p_j holds token or has granted access to another process

p_j votes for p_i to enter the critical region

Maekawa's Algorithm – Part 2

```
For pi to exit the critical section
   state := RELEASED;
   Multicast release to all processes in Vi;

On receipt of a release from pi at pj
   if (queue of requests is non-empty)
   then
      remove head of queue - from pk, say;
      send reply to pk;
      voted := TRUE;
   else
      voted := FALSE;
   end if
```

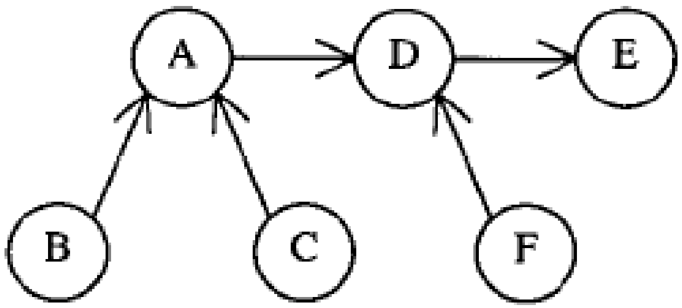
On leaving the critical region, all other processes in the set are informed

Now critical region is empty; give permission to next process in the queue

Raymond's Token-based Approach

Alternative to Maekawa's Algorithm:

- Processes are organized as an un-rooted *n*-ary tree.
- Each process has a variable HOLDER, which indicates the location of the access privilege relative to the node itself.



HOLDER_A	= D
HOLDER_B	= A
HOLDER_C	= A
HOLDER_D	= E
HOLDER_E	= self
HOLDER_F	= D

- Each process keeps a REQUEST_Q that holds the names of neighbors or itself that have sent a REQUEST, but have not yet been sent the privilege message in reply.

Raymond's Token-based Approach

To enter the critical region (CR):

- Enqueue self; if a request has not been sent to `HOLDER` before, send a request

Upon receipt of a `REQUEST` message from neighbor x:

- If *x* is not in queue, enqueue *x*
- If *self* is a `HOLDER` and still in the CR, it does nothing further
- If *self* is a `HOLDER` but exits the CR, then it gets the oldest requester from `REQUEST_Q`, sets it to be the new `HOLDER`, and sends `PRIVILEGE` to it

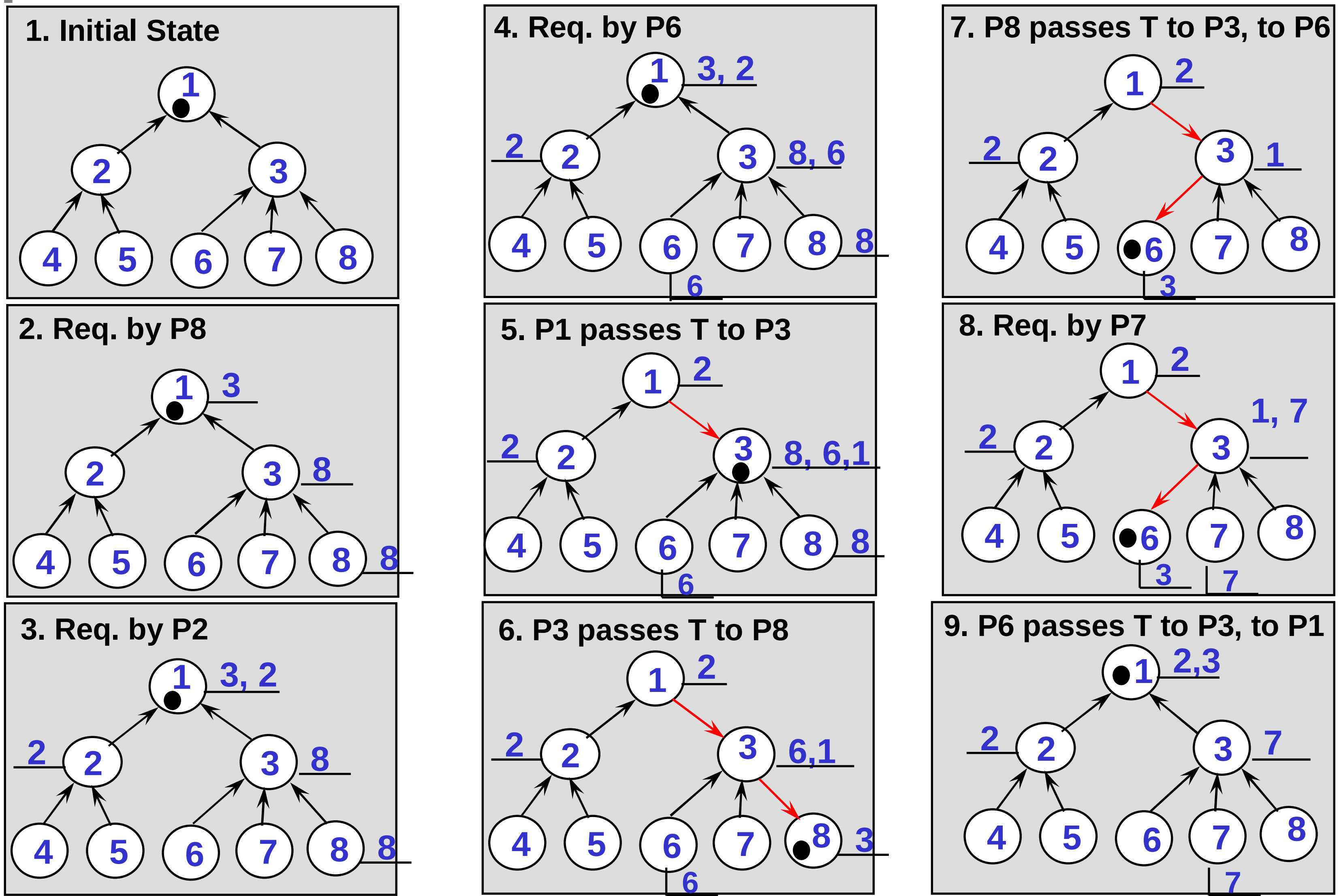
Upon receipt of a `PRIVILEGE` message:

- Dequeue `REQUEST_Q` and set the oldest requester to be `HOLDER`
- If `HOLDER` = *self*, then hold the `PREVILEGE` and enter the CR
- If `HOLDER` = *some other process*, send `PRIVILEGE` to `HOLDER`. In addition, if `REQUEST_Q` is non-empty, send `REQEST` to `HOLDER` as well

On exiting the CR:

- If `REQUEST_Q` is empty, continue to hold `PRIVILEGE`
- If `REQUEST_Q` is non-empty, then dequeue `REQUEST_Q`, set the oldest requester to `HOLDER`, and send `PRIVILEGE` to it. In addition, if the (remaining) `REQUEST_Q` is non-empty, send `REQUEST` to `HOLDER` as well

Example:Raymond’s Token-based Approach



Election Algorithms

Many distributed algorithms need a *coordinator/initiator/...*

- It does not matter which process takes over the special role
- Thus: *electing* a coordinator
- In general:
  - Locating the process with the best election value, very often the highest process number (network address, ...); it will become the coordinator
  - Any process can call for an election. The result of an election does not depend on the initiating process
  - There could be concurrent calls for the same election
- *Goal of election algorithm*

  To ensure that after an election started, all processes agree on the same process to be the coordinator
- Assumption of algorithms: each process knows the election values (in the following: the process numbers) of all processes, but it does not know which of the processes are up and running

Voting vs. Election

Difference between voting algorithms and election algorithms

- Voting is initiated to grant special rights (resource access) to any process
  - ➢ Processes are *not aware* of the result of their vote
  - ➢ Failures are *not considered*
- Elections are initiated to select a coordinator or to grant special privileges to a certain process
  - ➢ All processes are *informed* of the result
  - ➢ Election is usually initiated when process *failures occur*

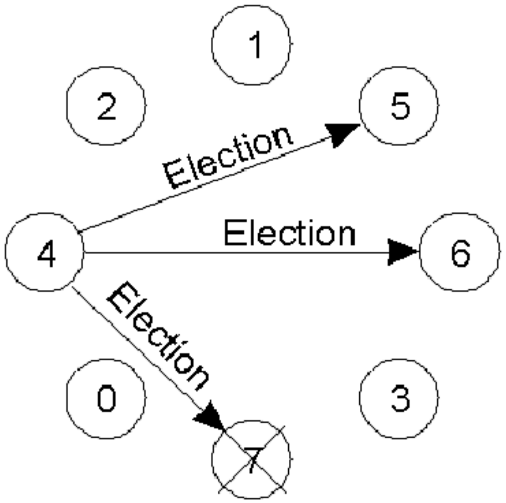
The Bully Algorithm

Bully Algorithm: when any process *P* notices that the current coordinator does not respond, it initiates the following steps:

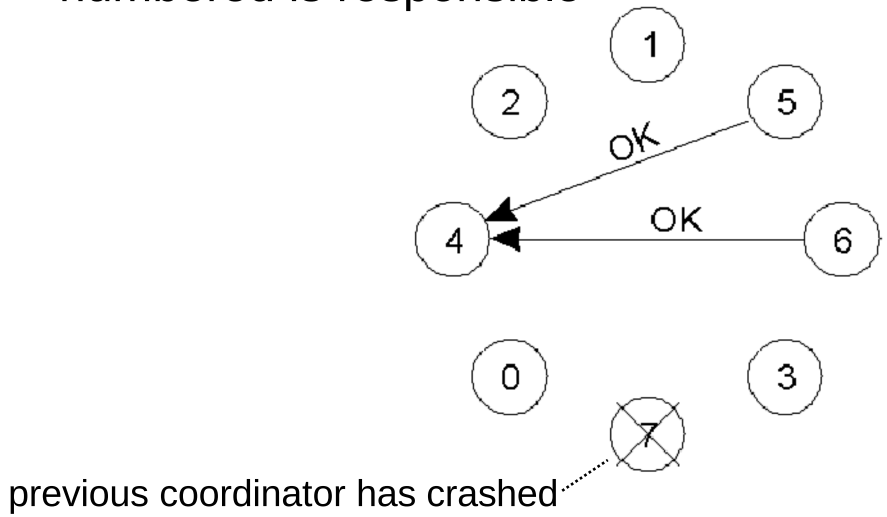
1. *P* sends an ELECTION message to all processes with higher numbers
2. If no one responds, *P* wins and becomes the new coordinator
3. If one of the higher-number processes answers, it takes over the election

Example:

- Process 4 mentions that Coordinator 7 is not responding. It sends an election message to all higher-numbered processes

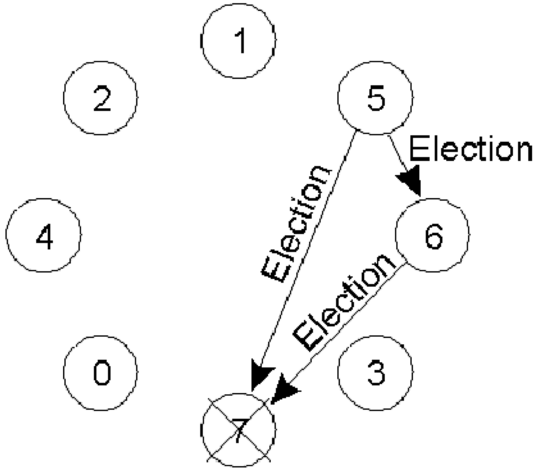


- The living processes 5 and 6 respond, 4 can stop his job. Someone higher-numbered is responsible

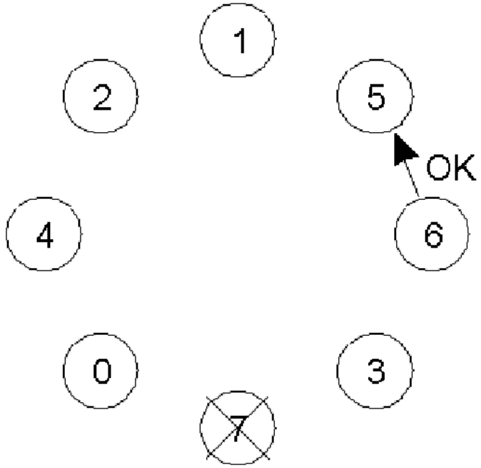


The Bully Algorithm

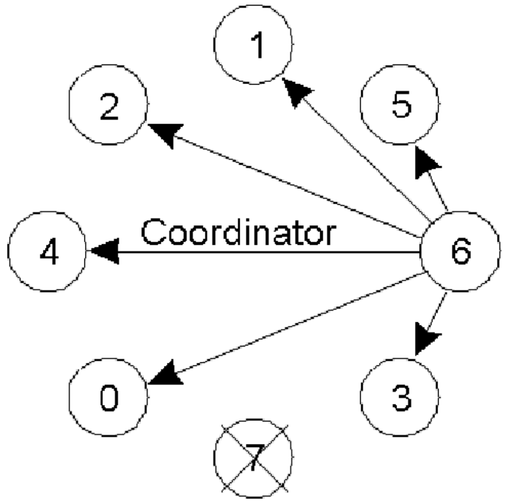
- Process 5 and 6 continue by sending election messages to all higher-numbered processes



- Process 5 receives an answer and can stop the election. Process 6 gets no answer (usage of timeouts for considering process failures) and knows that it is the highest-numbered living process



- Process 6 becomes the new coordinator and pushes this information to all processes



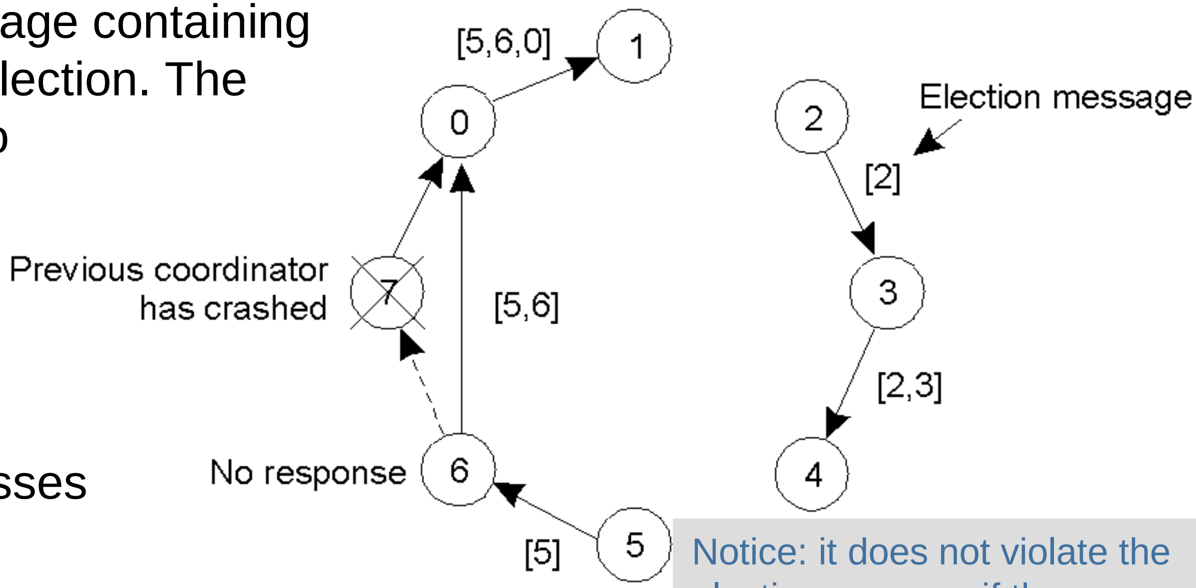
Note: if a process comes back which previously has been down, it starts an election because it does not know about the current coordinator

A Ring Algorithm

Alternative approach: seeing all processes as a ring (without token!)

Assumption: each process knows its successor

1. A process noticing that the coordinator is down sends an ELECTION message containing its process id to its successor
2. If the successor is down, the sender skips this process and looks for the next working process in the ring
3. Each process receiving the message adds its own process id to the message
4. A process receiving a message containing its own number, stops the election. The message type is changed to COORDINATOR and circulates ones more. The process with the highest number contained in the message is the new coordinator. All other processes are the new ring members.



Notice: it does not violate the election process, if there are two or more elections in parallel

Consensus Problems

Needed in some situations: Agreement of several processes on the 'correct' value of some data after one or more processes have proposed what the value should be (e.g. ‘go’ or ‘abort’)

Problems:

- A communication system never is completely reliable: loss/distort of messages
- Processes can be faulty or even malicious

Even in such cases, an agreement should be possible.

Usual example for explanation is the problem of **Byzantine Generals**:

Several byzantine generals surround a foreign camp and think about an attack. An attack can only be successful if all forces attack jointly. The generals exchange messages by (unreliable) riders. The riders can be catched (message loss), additionally some generals could be traitors (faulty processes).

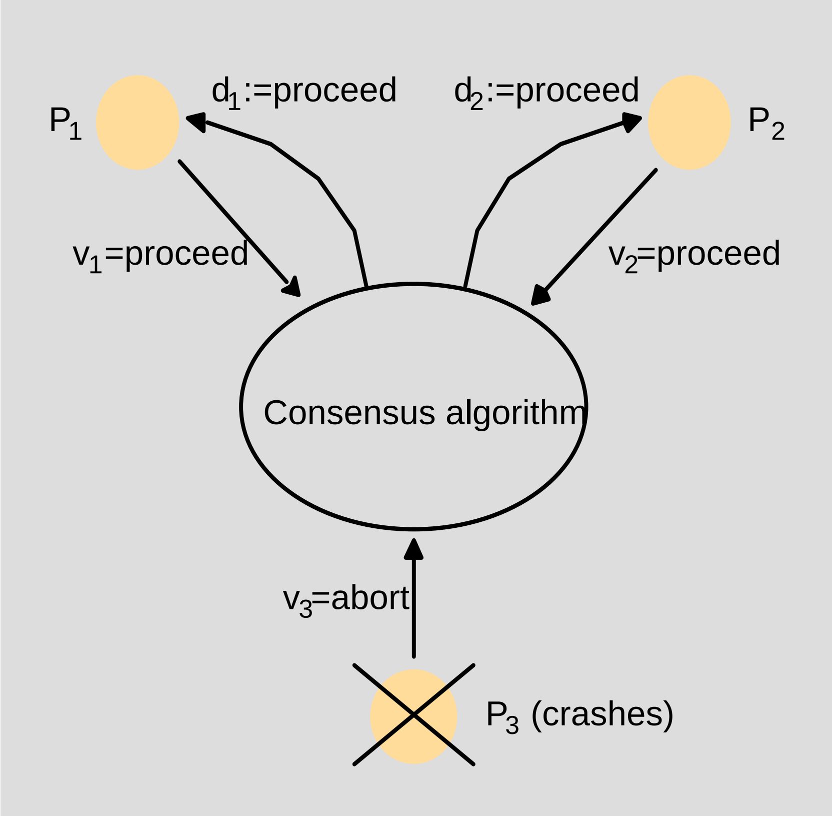
Definition of Consensus Problems

Every process p_i begins in a *undecided* state and proposes a value v_i from a set D

- The processes communicate with each other by exchanging their values
- Then each process sets a *decision variable d_i*. It enters the *decided* state in which values do no more change.

Requirements on a consensus algorithm:

- **Termination**: each correct process eventually sets the decision variable
- **Agreement**: the decision values of all correct processes are the same
- **Integrity**: if all correct processes propose the same value, this value is chosen in the *decided* state



If there are no faulty processes and message losses, the solution is simple: choose the value which was proposed by most processes.

Different Consensus Problems

General consensus problem

- Agree on a value fulfilling the requirements given above
- *Notice: solving consensus is equivalent to solving atomic broadcast*

Byzantine agreement

- Three or more generals have to agree to attack or to retreat
- One commander issues the order, the other generals decide to attack or to retreat
- Each of the generals (including the commander) can be 'faulty‘
- In this problem, the *integrity* requirement differs from the general formulation: if the *commander* is correct, then all correct generals agree on the value he had proposed

Interactive consistency

- Agreement on a *vector* of values, one for each participating process (*decision vector*)
- Now the *integrity* requirement is: if p_i is correct, then all correct processes agree on v_i as the i^{th} component of the vector

Consensus in a Synchronous System

Use atomic broadcast to implement consensus:

- Each process p_i proposes a value v_i
- p_i: `broadcast(A, v_i)`
- Each p_i chooses $d_i = m_i$ where m_i is the first value delivered to p_i
- Variant: collect all values and use another common strategy to choose one value

Or implement some new consensus algorithm

- Assumptions:
  - Processes p_i (i = 1, … N) have to reach consensus
  - Up to f processes can be faulty (crash failure only)
  - No byzantine failures
  - Synchronous system
  - Existence of basic broadcast assumed (only validity holds, no agreement)
- Basic idea:
  - Perform f+1 rounds to deal with all process failures
  - In each round, p_i broadcasts all new values it has not broadcasted before

Consensus in a Synchronous System

Process p_i: (`broadcast(B, x)` is basic broadcast of x)

initialization:

$\texttt{Values}_i^1 := \{v_i\};$ **Set of proposed values known to process p_i before round 1**

$\texttt{Values}_i^0 := \{\};$

In round r ($1 \le r \le f+1$)

$\texttt{broadcast(B, Values}_i^r \texttt{ - Values}_i^{r-1}\texttt{);}$

$\texttt{Values}_i^{r+1} := \texttt{Values}_i^r;$

`while (in round r)`

`{`

$\texttt{on deliver(B, V}_j\texttt{) from some p}_j$

$\texttt{Values}_i^{r+1} := \texttt{Values}_i^{r+1} \cup \texttt{v}_i;$

`}`

First step: broadcast all known values not sent before

Next step: collect all proposed values from other processes (synchronous system required to restrict duration of a round!)

After f+1 rounds

$\texttt{assign d}_i := \texttt{minimum(Values}_i^{f+1}\texttt{);}$ **Each process chooses the same value**

Consensus in a Synchronous System

Why does it work?

- Termination: given – limited number of rounds in a synchronous system
- Agreement and integrity: given if all correct processes end up with the same set of values

  Assume: the final sets of values of p_i and p_j differ: p_i possesses a value that p_j does not posses

  → there must be a p_k that crashed after sending v to p_i and before sending v to p_j (in round f+1)

  → Every process possessing v in previous rounds crashed

  → There has been at least one crash in each round, i.e. f+1 crashes (contradiction)

More complicated: Byzantine Generals

Given:

- *N* processes (generals) with at most *f* faulty processes (byzantine failures, including crash and omission failures)
- Private communication channels between pairs of processes
- Every message sent is delivered correctly, absence of messages can be detected (synchronous system!)

Goal: each correct process p_i computes a vector $x_i = (x_{i1}, x_{i2}, ..., x_{in})$ with

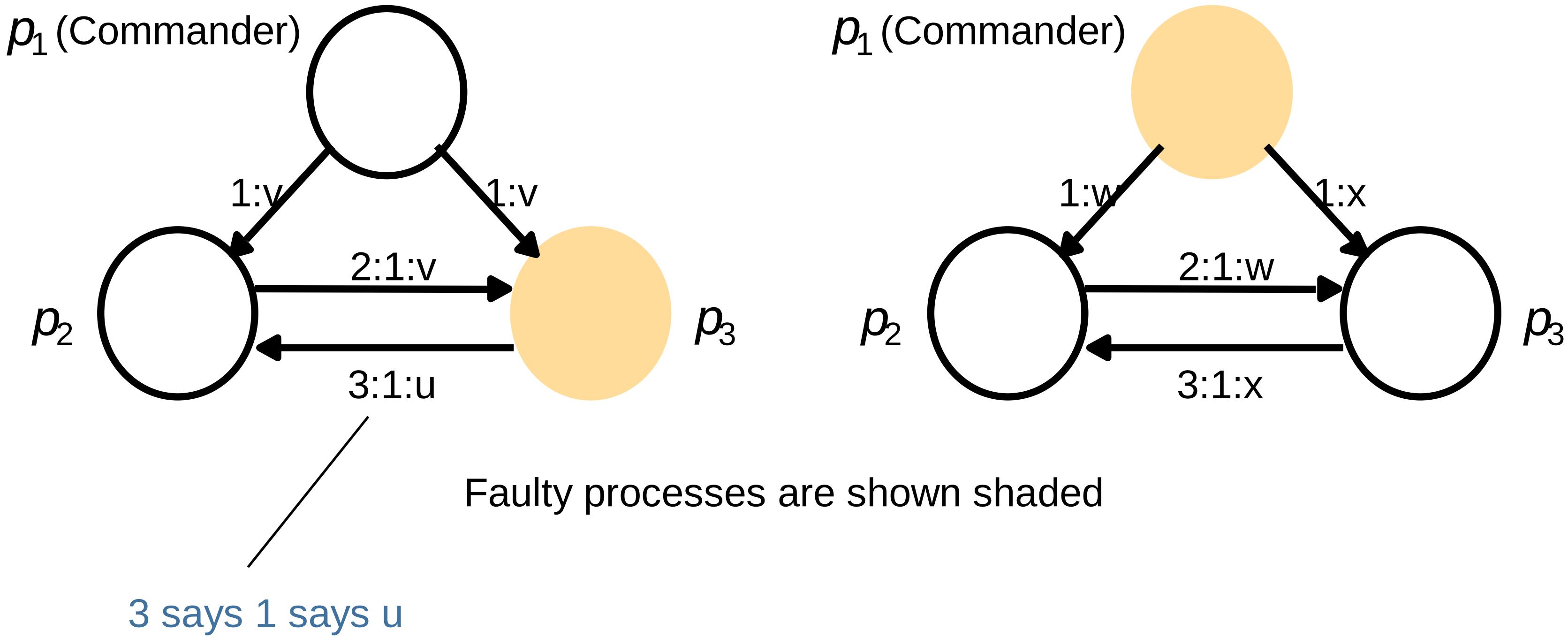
- $x_{ir} = v_r$, if p_r is correct
- $x_{ir} = x_{ik}$, if p_r and p_k are correct

All known solutions have the following characteristics:

1. An algorithm only terminates correctly if $N \geq 3f+1$
2. The worst case for agreeing is $f+1$ message delivery times
3. Large number of exchanged messages: each process has to collect all messages and execute the algorithm (it can not trust other processes)

Impossibility for *N* < 3*f*+1

Example: $N = 3$, $f = 1$



Faulty processes are shown shaded

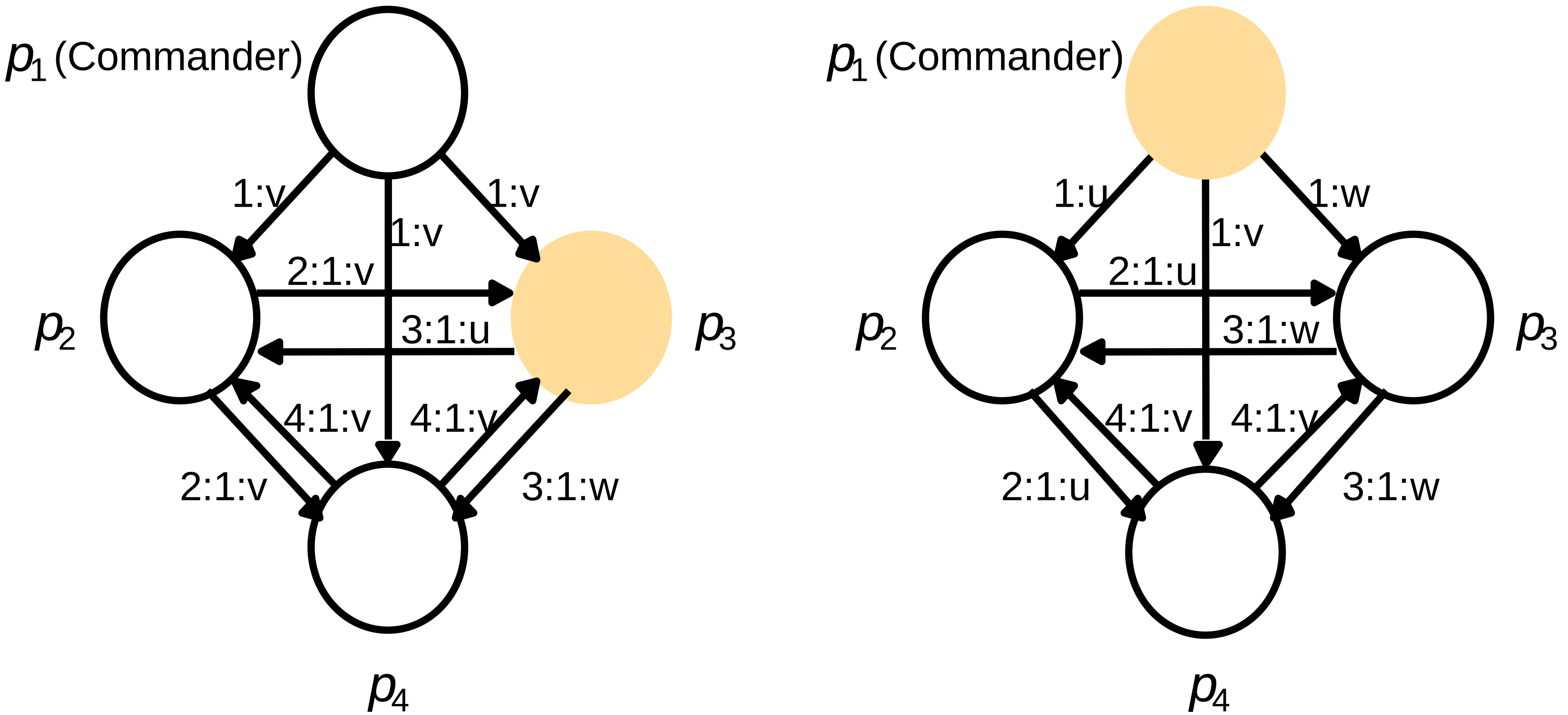
Two cases in which p_2 can not decide which value is correct

Example: N = 4

Simplest example for byzantine generals: N=4, f=1:

- Algorithm consists of two rounds
  - round 1: commander sends own value to the other three processes
  - round 2: other processes send values collected in the first round to the other processes
- Information of the faulty process can be wrong or even not send (then it can be set to a random value)
- Compute vector $x_i = (x_{i1}, x_{i2}, x_{i3}, x_{i4})$ [for correct process p_i]
  - $x_{ii} = v_i$
  - x_{ir} ($r \neq i$): at least two of the three incoming values are equal; set x_{ir} to this value (*majority decision*).

Four Byzantine Generals



Faulty processes are shown shaded

General Algorithm

universitätbonn

Assuming *f* faulty processes, we again need *f*+1 rounds

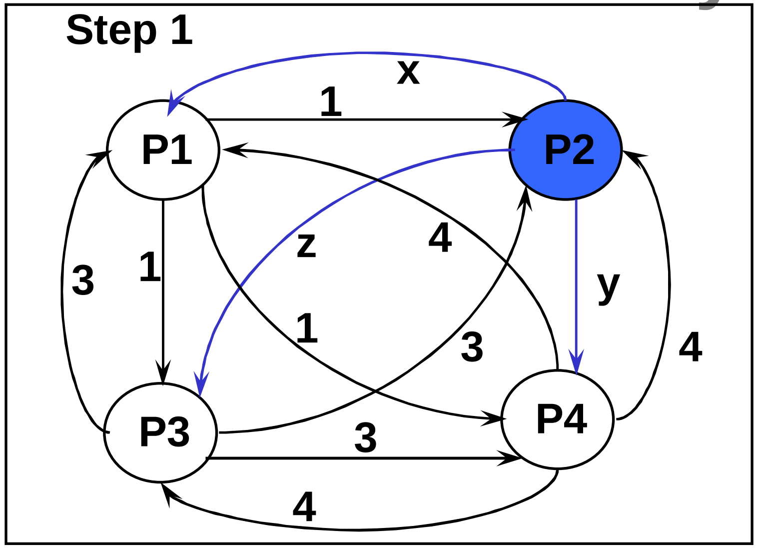
Algorithm BG(0,*S*)

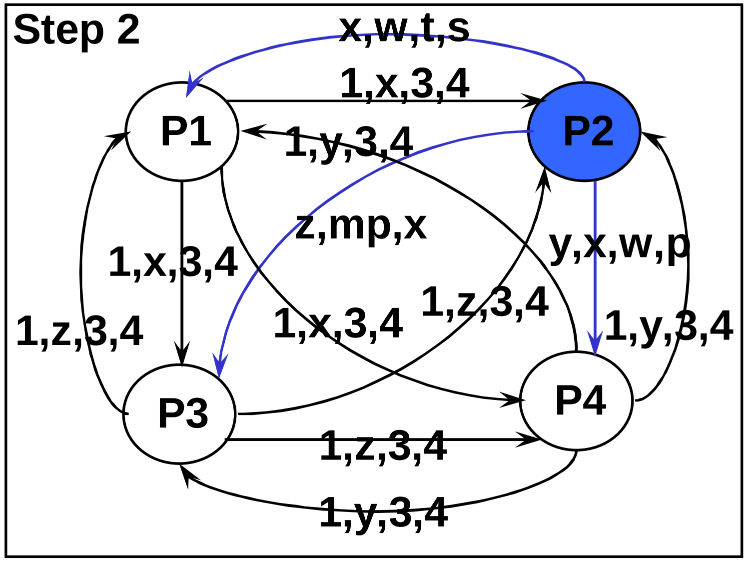
- The commander *i* sends his value *v* to every general $j \in S\backslash\{i\}$
- Every general $j \in S\backslash\{i\}$ accepts value *v* received from *i*

Algorithm BG(*f*,*S*) for *f* > 0

- The commander *i* sends his value *v* to every general $j \in S\backslash\{i\}$
- For each general $j \in S\backslash\{i\}$:
  - Let v_j be the value *j* received from commander *i*, or else be ⊥ if he receives no value
  - General *j* initiates algorithm BG(*f*-1,$S\backslash\{i\}$)
- For each general *j* and each $k \neq j$
  - Let v_k be the value general j received from general k in the former step (using BG(*f*-1,$S\backslash\{i\}$))
  - General *j* uses the value majority($v_m \mid m \in S\backslash\{i\}$)

Example: Byzantine Generals for Interactive Consistency





Step 3

P1 { 2: <x,w,t,s> 3:<1,z,3,4> 4: <1,y,3,4>	P2 { 1:<1,x,3,4> 3: <1,z,3,4> 4: <1,y,3,4>
P3 { 1:<1,x,3,4> 2: <x,w,t,s> 4: <1,y,3,4>	P4 { 1:<1,x,3,4> 2: <x,w,t,s> 3: <1,z,3,4>

Step 4

P1 <1,?,3,4> P2 <1,?,3,4>

P3 <1,?,3,4> P4 <1,?,3,4>

Impossibility Result

"No algorithm can guarantee to reach consensus in an asynchronous system, even with one process crash failure"

→ A proof for this statement exists!

→ Atomic broadcast thus also is impossible in asynchronous systems!

However, there are solutions

- Atomic broadcast: e.g. ISIS ABCAST
- Consensus: two-phase-commit, used within transaction processing
- … we just have to mask failures by recovery, or to use timeouts or something similar and assume that a process crashed (unreliable failure detection!)
- … and such solutions are much more efficient than consensus algorithms

Conclusion

When implementing distributed software, several services can be helpful to support the cooperation between the software components:

- Controlling access to shared resources – how to achieve mutual exclusion for shared resources
- Sometimes, one component in needed to become a coordinator – how to determine which component it should be
- How to come to an agreement between redundant components if fault tolerance is needed

→ Such services should be implemented by a middleware to give a basis for software development