Algorithms for distributed systems

- Overview
  - Distributed mutual exclusion
  - Election algorithms
  - Consensus algorithms

Coordination and matching algorithms

- Coordination and matching algorithms necessary in processes, where e.g.
  - coordination of activities regarding access to jointly used resources such as common objects
  - Agreement on a joint coordinator, for example if the current coordinator fails and need to be replaced $\Rightarrow$ Implementation of election algorithms in distributed systems
- Motivation
  - Complex devices with several information sources and controlling devices
    - Controlling devices must agree on a certain action, e.g. whether the currently running operation should be continued or aborted
  - Examples
    - Confirm or abort the start procedure of airplane on the runway
    - Activate the brakes in case of danger, e.g. in trains

Coordination and matching algorithms (2)

- Implementation approaches
  - Master node collects all information, evaluates the information and meets the decision
    - Straight forward implementation
    - Single point of failure, which can lead to crash of the entire system
  - Self-organizing process in a set of distributed, involved components
    - Complex implementation
    - Single point of failure avoided
- Examples for characteristic problems
  - Distributed mutual exclusion
  - Election algorithms
  - Consensus algorithms

Distributed mutual exclusion

- Critical sections: Chain of operations, where a concurrent processing may cause errors
- Examples
  - Object update (e.g. write access to a file): Lock the file during the first access, further write operation after write lock release by process currently updating the object
  - Peer-to-Peer coordination while using joint resources
    - Mobile ad-hoc networks $\Rightarrow$ solely one node is allowed to transmit on a jointly used channel
  - Real life: Number of free parking slots in a parking house with several entrances and exits
    - One process per entrance or exit
    - Common counter for the entire house
    - Consistent counter update by distributed processes necessary
  - We need an algorithm for distributed mutual exclusion without application of a master node

Mutual exclusion with semaphores

- Problem well known in system software
- Application of binary semaphores to implement mutual exclusion
  - Two states (locked, free) to protect critical sections
  - Critical section free, then access granted, otherwise the process is blocked (e.g. sleep()) and inserted into the queue assigned to the semaphore
- Solution not applicable to distributed systems, as no shared memory available ⇒ approach with messages needed

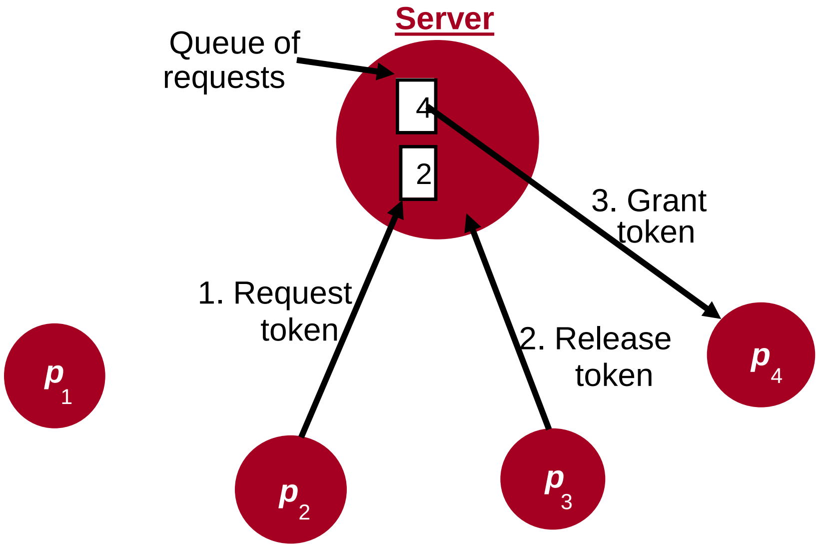
Mutex=Mutual Exclusion

Semaphore *Mutex*=1

Process *P1*		Process *P2*
↓		↓
P(*Mutex*)		P(*Mutex*)
}	Critical Section	{
V(*Mutex*)		V(*Mutex*)
↓		↓

Mutual exclusion with master and message passing

- Simplest solution: Using master node
  - Master as a superior unit grants or rejects access into the critical section
  - Process requests access by sending a request message to master node
  - Master queues all requests according to certain criteria (FCFS, priority, …) and sends reply message to the first element
  - Reply message contains a token as a ticket for the critical section
  - After leaving critical section, process sends the token to the master node and commits the critical section is free for the next process



Implementing mutual exclusion with master node

- Needed functions
  - Inter-process communication with send() and receive()
    - send(destination, &message)
    - receive(source, &message)
  - Indirect addressing using a data structure called mailbox = waiting queue for temporary message storage
  - Implementation with standard OS system calls
- Coordination procedure
  - Master queues all requests and sorts according to given criterion
  - First process receives reply message with the token
  - All other requests are blocked until the token is released
  - Then the next process in the queue receives the token

Implementing mutual exclusion with master node(2)

All processes use the mailbox *Mutex* for sending and receiving messages

```
const n=... number processes
main()
{
    createmailbox(mutex);
    send(mutex,&init);
    // Initial state with one token
 // Independent processes
        P(1);
        P(2);
        ...
        P(n);
    // in the distributed system
}
```

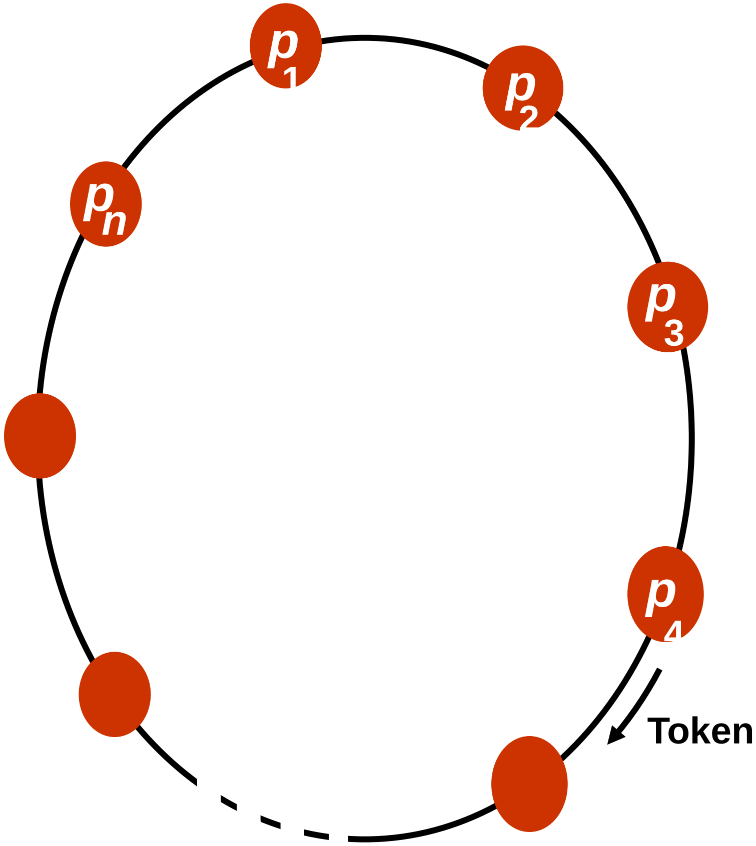
```
void P(int i) {
    message msg;
    while(TRUE) {
     receive(mutex, &msg);
    // Token received as
    // necessary prerequisite
    // to access critical section
     <critical section>;
     // Mailbox empty
     send(mutex,&msg);
     // Return the token to mailbox
     // ⇒ critical section free again
    }
}
```

Evaluation criteria

- Performance of an algorithm for mutual exclusion is evaluated according to following criteria
  - Used bandwidth: Proportional to number of sent messages
  - How long a client is deferred while waiting for entrance of exit from the critical section
  - System throughput
    - Efficiency regarding all processes requesting access to the critical section
    - Synchronization delay
- Example: Performance of the algorithm with a master node
  - Two messages needed to enter the critical section, even in case of a free critical section $\Rightarrow$ Delay equals the round trip time for messages
  - Server as bottle neck

Ring based algorithm

- Mutual exclusion between N processes without additional master process
  - ➢ Processes arranged in a logical ring structure
  - ➢ Process Pi with communication channel to the next process P(i+1) mod N
- Approach
  - ➢ Entry token circulates in the ring
  - ➢ Processes without entry request pass the token to their neighbor
  - ➢ Process requesting entry keep the token and enter the critical section. After leaving the critical section, the token is passed to the next process in the ring
- Evaluation
  - ➢ High bandwidth consumption
  - ➢ Entry delay: 0 … N messages
  - ➢ Synchronisation delay: 1 … N messages



Multicast algorithm with logical time stamp

- Considering peer processes
  - Process requesting entry sends multicast message (to all other peers)
  - Access granted, if and only if all other peer processes reply to the request
- Prerequisites
  - Each process Pi has a counter with logical time
  - Messages include tuples <T,Pi> with T as time stamp and Pi as sender ID
  - Each process holds the current status in the variable state
    - RELEASE: outside the critical section, no entry requested
    - WANTED: entry requested
    - HELD: process already entered the critical section
- Procedure
  - On request message, all processes test their status: if all processes in state RELEASE, the processes reply immediately to the request ⇒ access granted
  - At least one process in state HELD ⇒ no reply ⇒ access temporarily rejected

Multicast algorithm with logical time stamp (2)

- (Nearly) simultaneous requests
  - Two or more processes
  - request access ⇒ process with smaller time stamp receives all N-1 replies first ⇒ Access granted
  - Identical Lamport time
  - total ordering using PIDs
- Pro
  - Synchronisation delay of transfer time for a single message
- Con
  - Expensive algorithm regarding bandwidth

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 (nr 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;

Leader election algorithms

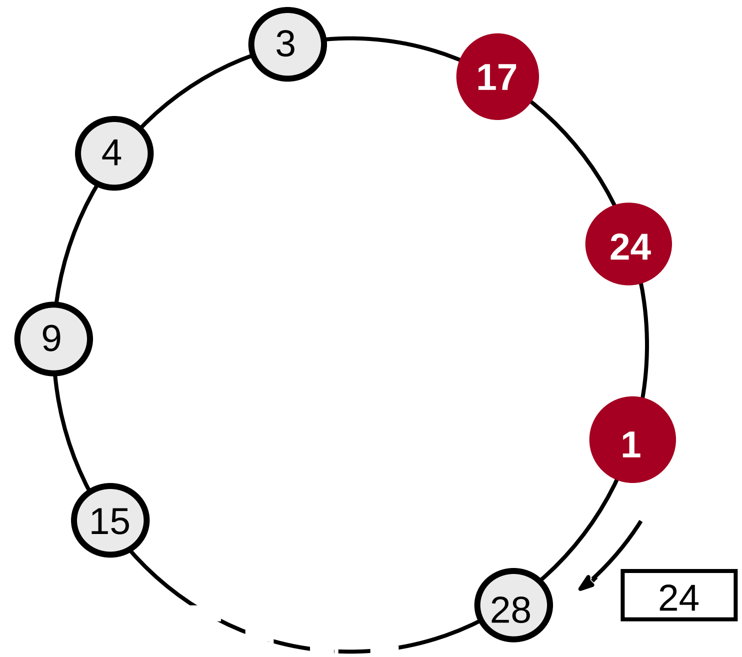
- Goal: elect a leader that will take over a certain function
  - Elect a (replacement) master node
  - Important: all nodes accept the decision
- Basic assumptions
  - Each participant has a unique identifier
  - Goal is to choose that member with the largest identifier as leader
    - Set of all identifiers unknown to all participants
- Fault assumptions
  - Processes may or may not fail, may behave in a hostile fashion
  - Messages may or may not be lost, corrupted, …
  - Different algorithms can handle different fault assumptions
- Time assumptions
  - Synchronous time model – all processes operate in lock-step, bounded message transit time?
  - Asynchronous model – no such bounds available?

Leader election algorithms – Concurrency

- Assumptions
  - Any process is allowed to start an election $\Rightarrow$ but one process is allowed to start only one election process at time
  - N processes can start N concurrent elections $\Rightarrow$ the final decision must be consistent despite the concurrency
  - Process Pi is at any time either a participant of the election or the process is not involved in the election
  - Each process receives a unique identifier ID$\Rightarrow$ process with the largest ID wins the election
  - Each process has a variable elected , where the ID of the elected process is stored $\Rightarrow$ Special symbol denotes an undefined value, e.g. election not finished yet

Ring-based election algorithms

- Available for processes arranged in a logical ring
  - ➢ Each process has a channel to his direct neighbors
  - ➢ Messages are sent in one direction, asynchronously
- Goal: Elect the processes with the largest ID as coordinator
- Initial state
  - ➢ Process 17 starts the election, changes the variable state to participant, sends an election messages with his own ID
  - ➢ Each receiving process compares the ID contained in the message with his own ID



Ring-based election algorithm (2)

- Result of the comparison
  - Received ID larger than the own ID ⇒ change status to participant, pass the message unchanged to the next neighbor
  - Received ID smaller than the own ID and status "no participant" ⇒ change status to participant, insert own ID in the message, pass the message to the next neighbor
  - Received ID identical with own ID (and already participant) ⇒ current process has the largest ID ⇒ Election finished ⇒ Coordinator set the variable to no participant and gives his ID in an elected message to the next neighbor
  - Receiving processes change status to "no participant", note the coordinator and pass the message to the next neighbor
- A process has to receive its own ID before sending elected messages
- Important: unique IDs, so two elected processes also in case of concurrency not possible
- Worst-case performance: 3N-1 messages
- Communication failures or node crashes stop the entire algorithm ⇒ little use in real world applications

Bully algorithm

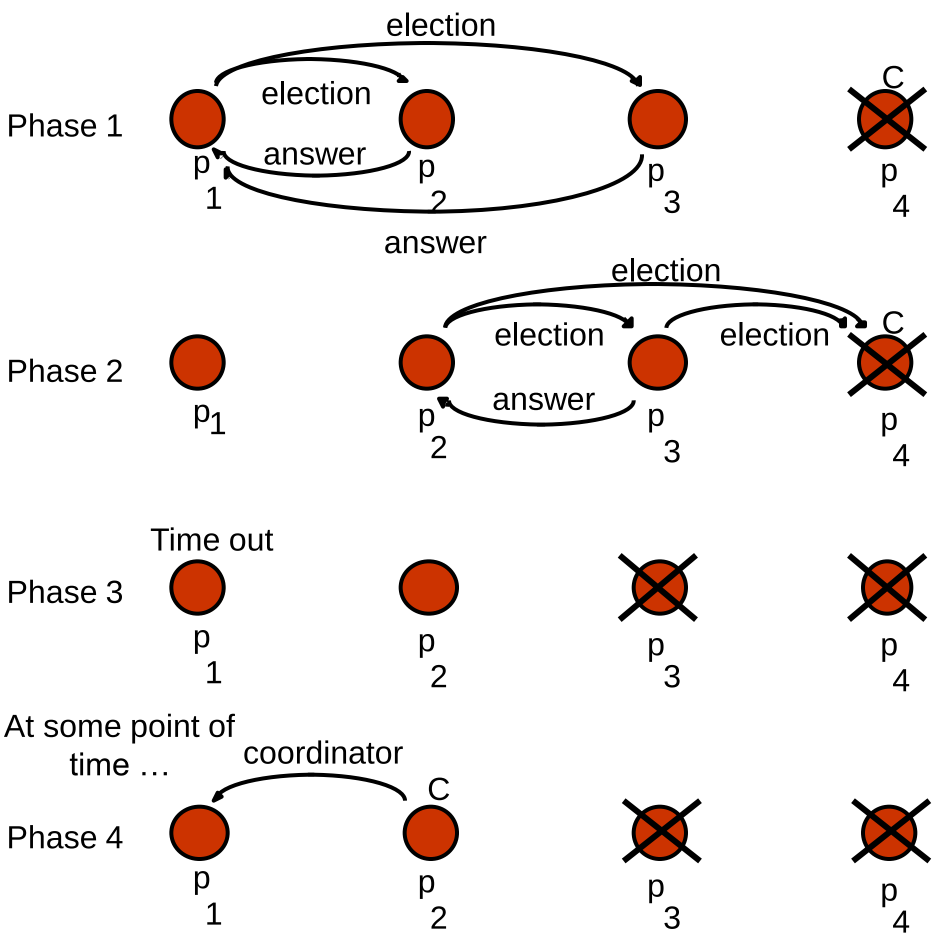
- Elect a process from P1, P2, …, PN with the largest ID as coordinator
- Prerequisites
  - Reliable communication, individual processes may crash
  - Synchronous communication $\Rightarrow$ Node failure detection with time outs
  - Each process knows all processes with larger ID than its own and is able to communicate with those
- Used message types
  - Message election: call for election
  - Reply message answer: reaction to election call
  - Coordinator message: call of the elected coordinator
- Setting time outs
  - Estimate the maximal transmission delay Ttrans and the maximal processing delay Tprocess
  - Upper bound T=2Ttrans + Tprocess
  - Time out expired $\Rightarrow$ notify node failed

Bully algorithm: procedure

- Process with largest ID elects itself to coordinator
- Process Pi with non-maximal ID calls for election
  - Election message to all processes with higher IDs
  - Wait for answer during time T
    - No message received $\Rightarrow$ the process that called the election declares itself to coordinator and sends the coordinator message
    - Receives a process Pj with IDj>IDi a message, then the process sends a reply message to Pi. Pi has to wait for an additional interval T´ for the coordinator message. If no message received during this time span, a new election is started
  - Process Pj starts a new election and repeats the workflow until the process with the largest ID was determined
- If the former, temporarily failed coordinator is restarted, it starts a new election. If the process still has the largest ID, then this process will be elected as coordinator again

Bully algorithm: Example

- Process P1 detects coordinator (P4) failure and starts an election (Phase 1)
- P2 and P3 send a reply answer to P1 and start their own elections
- P2 receives reply from P3, P3 receives no replies ⇒ P3 has the largest ID and declares itself to coordinator (Phase 2)
- Assume: Before P3 is able to send the coordinator message, P3 fails
- P1 waits until interval T´ expired without receiving a coordinator message ⇒ New election is started
- P2 elected to coordinator

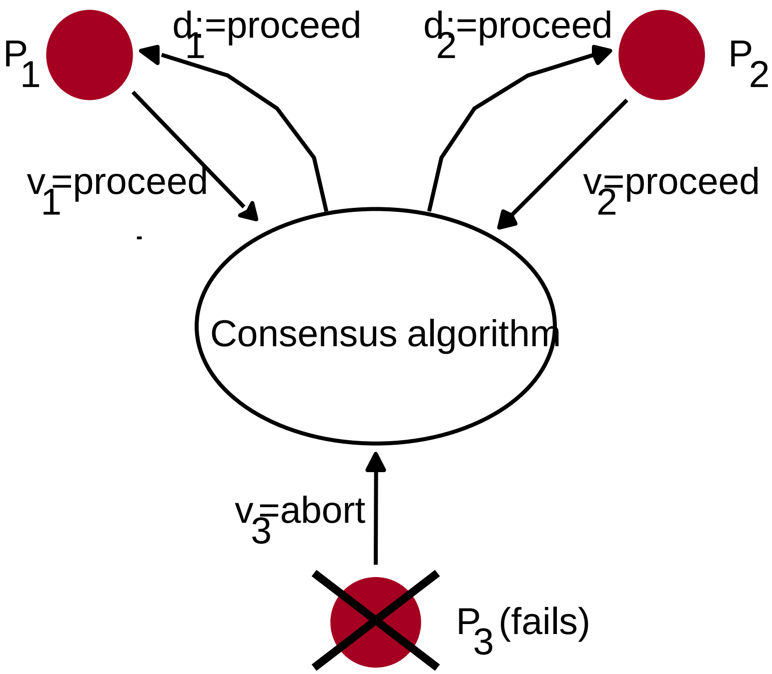


Consensus algorithms

- Consensus decision-making = group decision making process that not only seeks the agreement of most participants, but also to resolve or mitigate the objections of the minority to achieve the most agreeable decision
  - Redundant devices agree on GO or NO GO
  - Bank transfer: involved nodes agree to commit the same amount on both accounts (negative and positive)
  - Entering a critical section $\Rightarrow$ Decision on elected process
- Number of specialised protocols for sub-problems exists
- Looking at the general problem and solution approaches
- Possible failure sources
  - Errors during communication $\Rightarrow$ Loss or manipulation of messages
  - Failed processes lead to unpredictable behavior
    - Fail-stop failure: Process stops $\Rightarrow$ Failure discovery with time-outs
    - Byzantine failure: process delivers wrong results $\Rightarrow$ incorrect messages are produced $\Rightarrow$ Threat for the integrity of the entire system

Definition

- Each process Pi is in an undefined state and votes for a decision vi, which is part of the set with all possible decisions D (i=1,2,…,N)
  - Processes exchange messages with their votes
  - Decision based on the selected algorithm
  - Each process stores the agreed value in the decision variable di
  - Change of condition in dedicated state, di must not be changed any more
- Example
  - 2 processes vote for proceed, 1 process votes for abort
  - Last process crashes ⇒ decision for proceed



Consensus for three algorithms

Demands

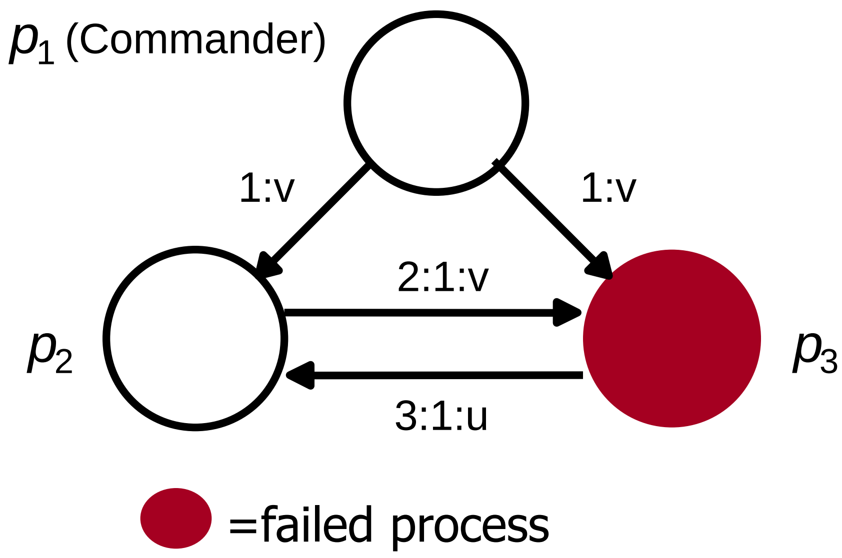
- For correct execution of a consensus algorithms, following must be fulfilled
  - Termination: Each correct process sets the decision variable at some point of time
  - Agreement: decision value is identical for all correct processes: if Pi and Pj are in dedicated state, then di=dj (i,j = 1,2, …, N)
  - Integrity: If all correct processes voted for the same value, then each process adopted this value in the dedicated state
- Problems?
  - Consensus in fail-safe systems: wait, until all votes arrived – including the own vote – and analyse the majority ⇒ In worst case, no majority can achieved and the state remains undefined ⇒ Additional decision measures necessary
  - Failures possible ⇒ not sure that all votes arrive or arrive correct
    - Reaction and termination of the consensus algorithm must be guaranteed

Example: Byzantine Generals' Problem

- Agreement problem
  - Generals of the Byzantine Empire's army must decide unanimously whether to attack some enemy army
  - Problem is complicated by
    - Geographic separation of the generals, who must communicate by sending messengers to each other
    - Presence of traitors amongst the generals.
  - Traitors can act arbitrarily in order to achieve the following aims
    - Trick some generals into attacking $\Rightarrow$ Force a decision that is not consistent with the generals' desires, e.g. forcing an attack when no general wished to attack $\Rightarrow$ failed processes
    - Catch a messenger $\Rightarrow$ some generals are not able to make their mind $\Rightarrow$ failed communication channels
  - If the traitors succeed, any resulting attack is doomed, as only a concerted effort can result in victory

Byzantine Generals in a synchronous system

- Simple variant: order by commander transmitted to other generals
- Prerequisites
  - Up to f of N processes may fail or produce false results ⇒ failed process can send a message with any value (also wrong values) any time
  - Missing messages recognized via time-outs
  - Private communication channels ⇒ other processes can't read the transmitted data
- Impossible mission in case of three available processes ⇒ if one of the processes fails, no solution possible (Extension for N≤3f)

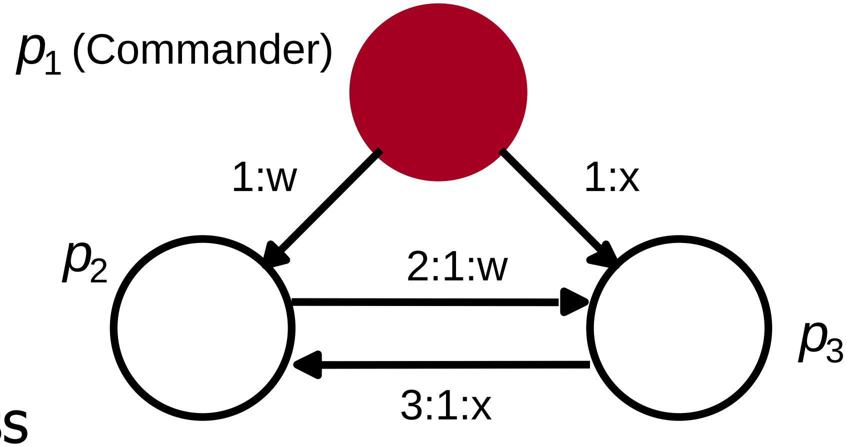


- Scenario
  - Prefix = Message source
  - : = „says“, 3:1:u = 3 says 1 says u
  - Commander gives the same order to both sub-commanders, P3 transmits a wrong order

  ⇒ P2 is not able to detect whether the order by the commander or by the sub-commander P3 is wrong

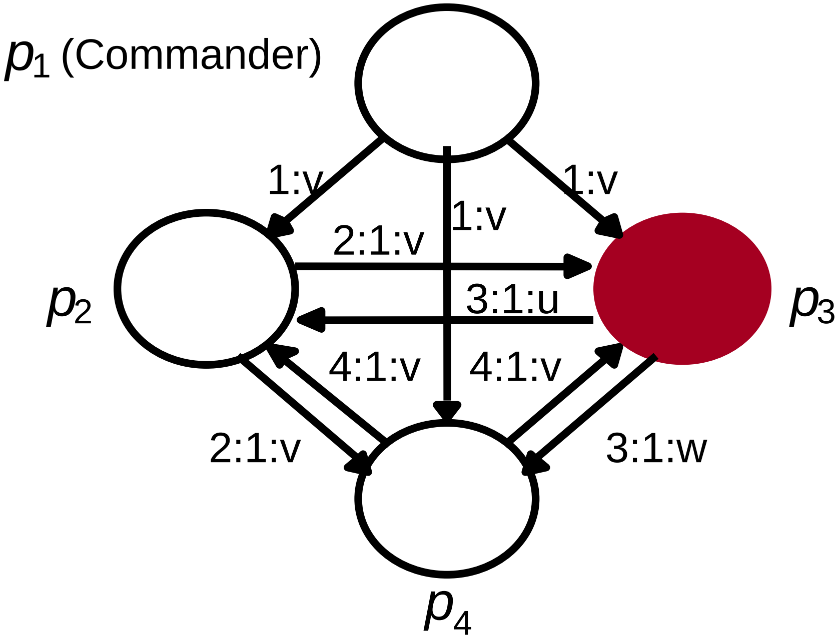
Byzantine Generals in a synchronous system (2)

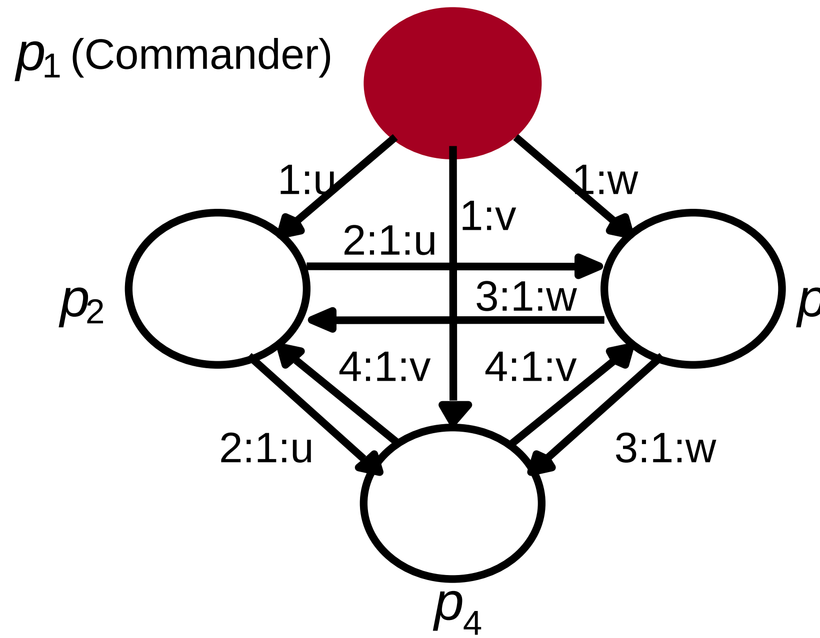
- Scenario: Commander failed
  - Different values to sub-commanders
  - P2 receives different values
  - ⇒ Same situation as in case of failed P3



- Solution for N = 4 and 1 failed process
  - Correctly working commanders have an agreement in two message passing intervals
    - Commander sends the value to each sub-commander
    - Each sub-commander sends the received value to the peered sub-commanders
  - Each sub-commander receives the value from the commander and N-2 values from peered sub-commanders
    - Commander failed ⇒ all sub-commanders may have the same value set, no failure is recognized, or they may have different values ⇒ failure detected
    - One of the sub-commanders failed ⇒ Detection by different value sets

Example for 4 sub-commanders





- Failed sub - commander
  - P2 determines majority(v,u,v)=v
  - P4 determines majority(v,v,w)=v
- Both processes detect failed sub-commander

- Failed commander
  - P2 determines *majority*(*u*,*v*,*w*) = $\perp$
  - P3 determines *majority*(*u*,*v*,*w*) = $\perp$
  - P4 determines *majority*(*u*,*v*,*w*) = $\perp$
- $\perp$ = special symbol, no majority possible

Byzantine Generals in asynchronous system

- These is no algorithm that guarantees consensus in asynchronous system, if processes may fail
  - Processes can answer to messages at any point of time $\Rightarrow$ differentiation between slow/delayed and failed processes not possible
  - Proof for non- existence of such an algorithm by Fischer et al.
- Approaches for work-around
  - Using partially synchronous algorithms: relaxed synchronous distributed systems, but still with well-defined upper and lower bounds
  - Masking faults: process data is stored persistently and restored in case of process failure $\Rightarrow$ hiding problems by controlled multiple execution, e.g. in case of transactions
  - Failure detectors: Processes agree, if some of them did not answer for a certain time and send no keep alive signal, then they are declared failed and are removed from the future decision process