Coordination & Agreement

CDK 15

MDS E2015
Søren Debois

Meta

MDS/MODIS Evaluations 2015

	Average	n	6	5	4	3	2	1
BSWU/MODIS								
Course	5,44	39	22	12	5			
Søren	5,56	36	23	11	1	1		
Holger	5,46	13	8	3	2			
Frederik	5,71	14	11	2	1			
KSDT/SMDS								
Course	4,33	3		1	2			
Søren	5,33	3	1	2				
Holger		0						
Frederik		0						

Comments

- You like:
  - TAs & TA support for mini-projects
  - Lecture contents & structure
  - Mini-project/lecture alternation

Comments

- You dislike:
  - Too few lectures
  - Too large groups
  - My articulation

Evaluation summary

- The course is functioning
- Next year: better group sizes
- I’ll keep working on my speech
- What do you think?

Recap

- P2P Motivation: Resource sharing.
- Unstructured networks (flooding, random walk)
- Structured networks (routing tables)
- Example: Pastry (ring-topology)
- Security (trust!)
- Applications

Coordination & Agreement

Plan

- Motivation: Fundamentals of agreeing.
- Distributed Mutual Exclusion
- Elections
- Consensus

Motivation

Redundancy

The Space Shuttle had 5 redundant avionics computers.

- Fundamental engineering problem: Making computers agree on anything.

- Applications everywhere in engineering, protocol design, P2P, …

- Fundamental problem of computer science: What can we/can we not do with computers?

Distributed Mutual Exclusion

- Prevent distributed processes from simultaneously “doing something” (be in a critical section).
- E.g., don’t issue “fire” and “move” simultaneously to your automated cannons.
- E.g., move operation on distributed file system.
- Like mutual exclusion in thread-programming.

- Process p_i among $p_1 \dots p_n$
- Asynchronous, reliable message delivery, no process failures.
- API:
  enter()
  resourceAccess()
  exit()

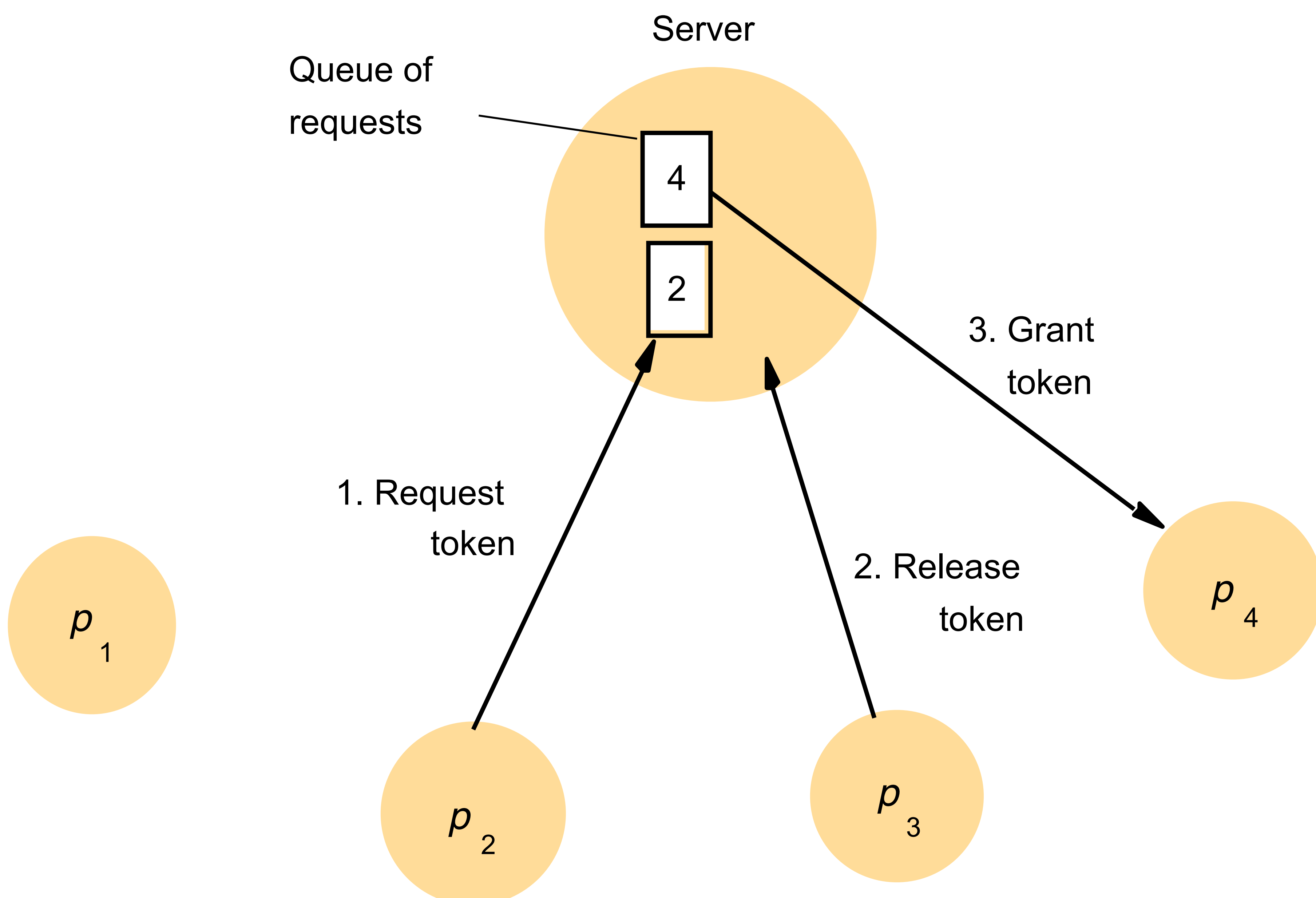
Discuss:
Design a protocol for achieving distributed mutual exclusion.

- Properties:
  ME1 (safe): at most one process in CS at any time.
  ME2 (live): request to enter/exit eventually succeeds.
  ME3 (order): entry to CS respects happens-before of enter() calls

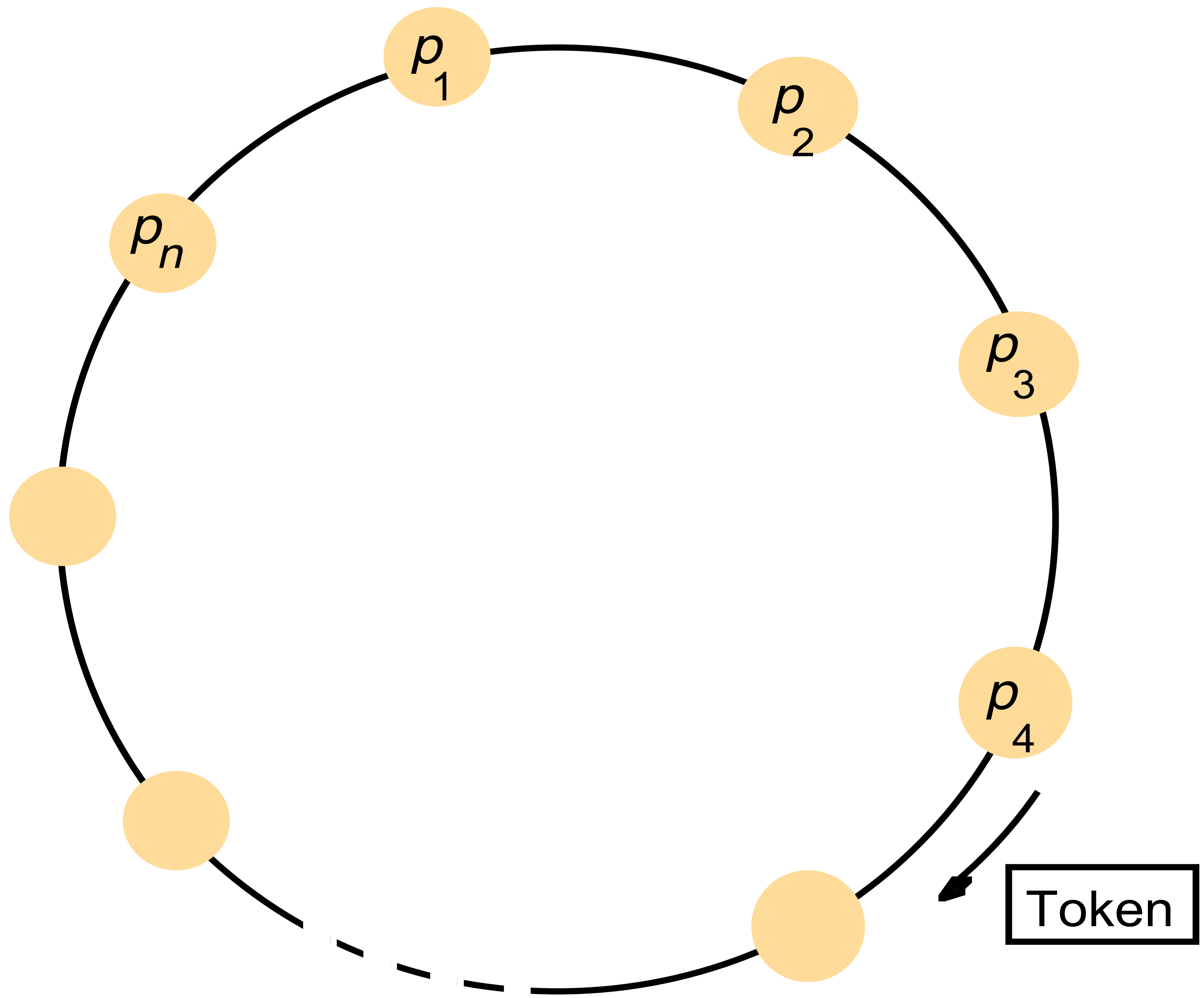
Solution 1: Central server.

Duh.

Solution 1: Central server.



Solution 2: Token ring.



Solution 3: Ricart & Agrawala

- On initialisation:
  state := RELEASED;

- On enter():
  state := WANTED;
  “multicast ‘req’”
  T := time of ‘req’
  wait for N-1 replies
  state := HELD;

- On recv ‘req (T_i,p_i)’:
  if(state == HELD ||
  (state == WANTED &&
  $(T,p_{me}) < (T_i,p_i)$))
  then queue req
  else reply to req

- On exit():
  state := RELEASED
  reply to all in queue

Comp

Discuss:
Compare central server, ring, Ricart & Agrawala,

- Satisfies ME3 (order)?
- Bandwidth consumed. Approximated by “number of messages sent”.
- Client delay for entry()/exit() operations.
- Throughput. Approximated by synchronization delay (time between exit()/enter()).

Comparison

	Server	Token-ring	Ricart & Agrawala	
Bandwidth	2 1	(continuous)	2(N-1) N-1	enter() exit()
Client delay	2 (rt) 0	0…N (avg .5N) 1	2(N-1) 0	enter() exit()
Sync delay	2 (rt)	1…N (avg .5N)	1	

rt = roundtrip time

Fault-tolerance

- Neither works in the presence of crash failures, lost messages.

Summary:
Distributed mutual exclusion

- Correctness: safe, live, (ordered).
- Algorithms: server, token ring, Ricart & Agrawala
- Criteria: bandwidth, client delay, sync delay.

Elections

The problem

- We want to elect a mutual exclusion server from a set of peers.
- Any process may *call the election*. There may be concurrent such calls.

The problem, in detail

- Process p_i among $p_1 \ldots p_n$
- Each process p_i has a variable $elected_i$.
- We require that
  E1 (safety): Always $elected_i = P$ or $elected_i = \perp$
  E2 (liveness): Correct processes set $elected_i \neq \perp$

Solution: Ring

Initialisation of p_i:
  state := non-participant
  $elected_i$:= ⊥

Call election by p_i:
  send ‘election(i)’

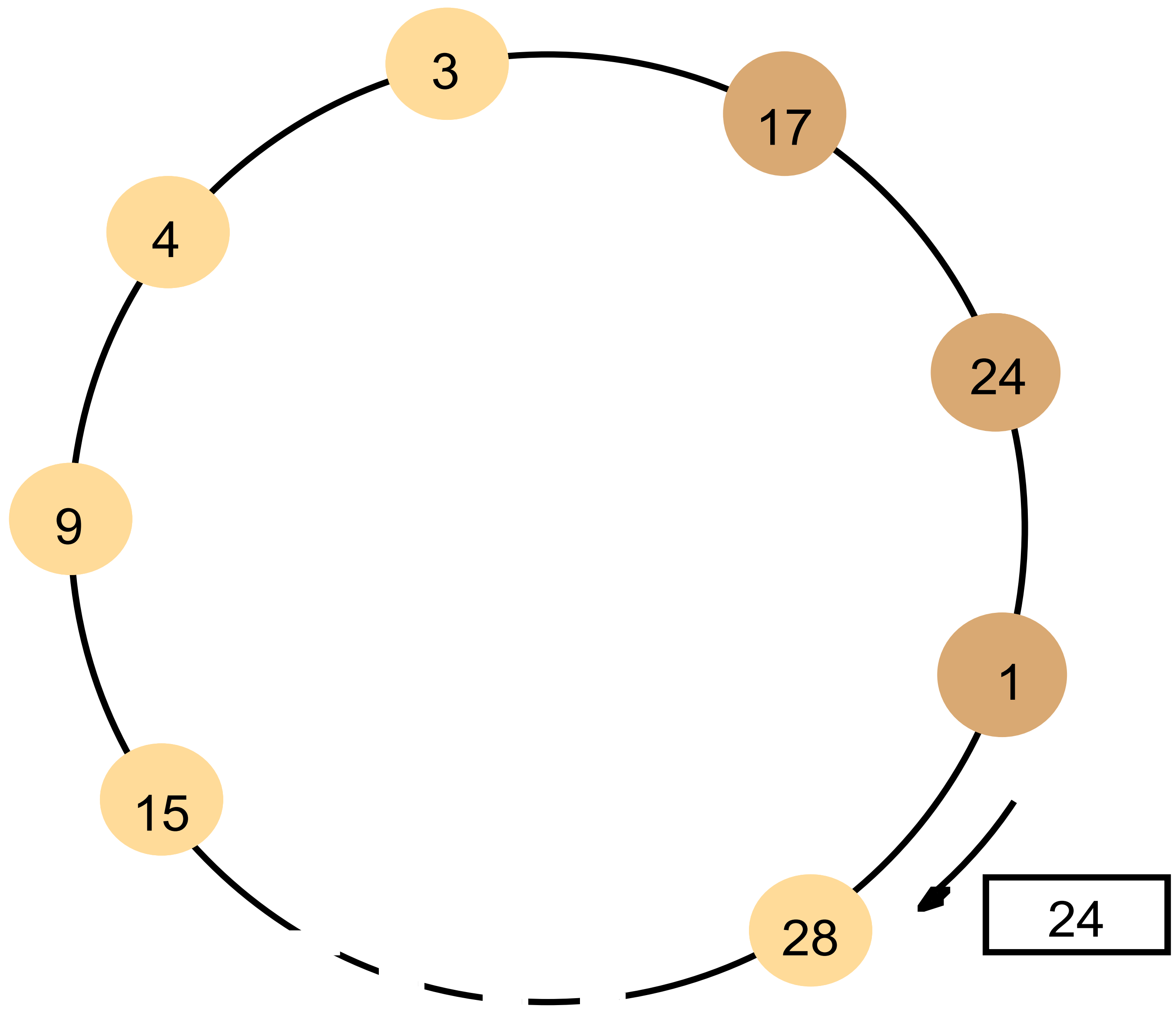
On receive ‘elected(j)’:
  if (i == j) then
    /* Do nothing. */
  else
    $elected_i$:= j
    send ‘elected(j)’

Solution: Ring

```
On p_i receiving 'election(j)':
  if (j == i) then
    elected_i = i
    send 'elected(i)'
  else if (j > i) then
    send 'election(j)'
  else if (state == non-participant) then
    send 'election(i)'
  else
    /* Do nothing */

  state := participant
```

Figure 15.7
A ring-based election in progress



Note: The election was started by process 17.
The highest process identifier encountered so far is 24.
Participant processes are shown in a darker colour

Correctness?

```
On pᵢ receiving ‘election(j)’:
  if (j == i) then
    electedᵢ = i
    send ‘elected(i)’
  else if (j > i) then
    send ‘election(j)’
  else if (state == non-participant) then
    send ‘election(i)’
  else
    /* Do nothing */

  state := participant
```

Evaluation

- Bandwidth (no. messages sent)
- Turn-around (max no. sequential messages in run)

Evaluation

- Bandwidth (no. messages sent):
  Worst-case, single election: 3N-1

- Turn-around (max no. sequential messages in run):
  Worst-case, single election: 3N-1

Tolerates no failures.

Solution 2: Bully

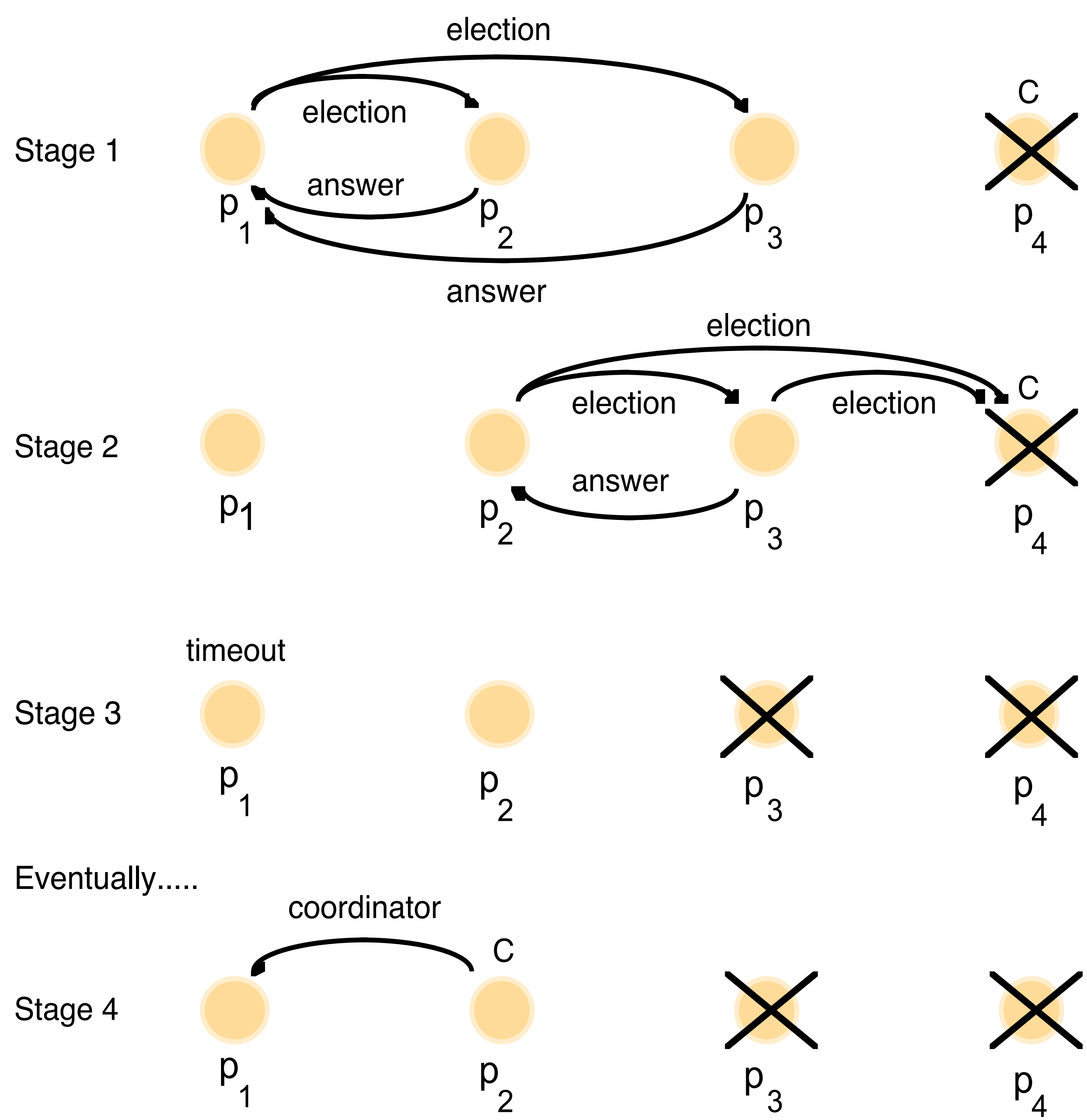
- Synchronous dist. sys.; crash failures allowed; detect crash failures by timeouts.
- Fixed set of initial processes known to all.
- Messages: election, answer, coordinator

Solution 2: Bully

- Call election by sending to higher-id processes **election.**
- Highest id process m declares himself winner by sending **coordinator(m)** to all.
- Others sends back **answer**, then call new elections.
- If timeout on caller c without **answer**, all higher ids must be gone; caller now have highest id and sends **coordinator(c)** to all.
- If timeout on caller c without **coordinator,** highest-id process must be gone; call new election.
- A new process with maximum id m immediately sends **coordinator(m)** to all.

Figure 15.8
The bully algorithm

The election of coordinator p_2, after the failure of p_4 and then p_3



Bully Correctness

- Call election by sending to higher-id processes **election.**
- Highest id process m declares himself winner by sending **coordinator(m)** to all.
- Others sends back **answer**, then call new elections.
- If timeout on caller c without **answer**, all higher ids must be gone; caller now have highest id and sends **coordinator(c)** to all.
- If timeout on caller c without **coordinator,** highest-id process must be gone; call new election.
- A new process with maximum id m immediately sends **coordinator(m)** to all.

Summary: Elections

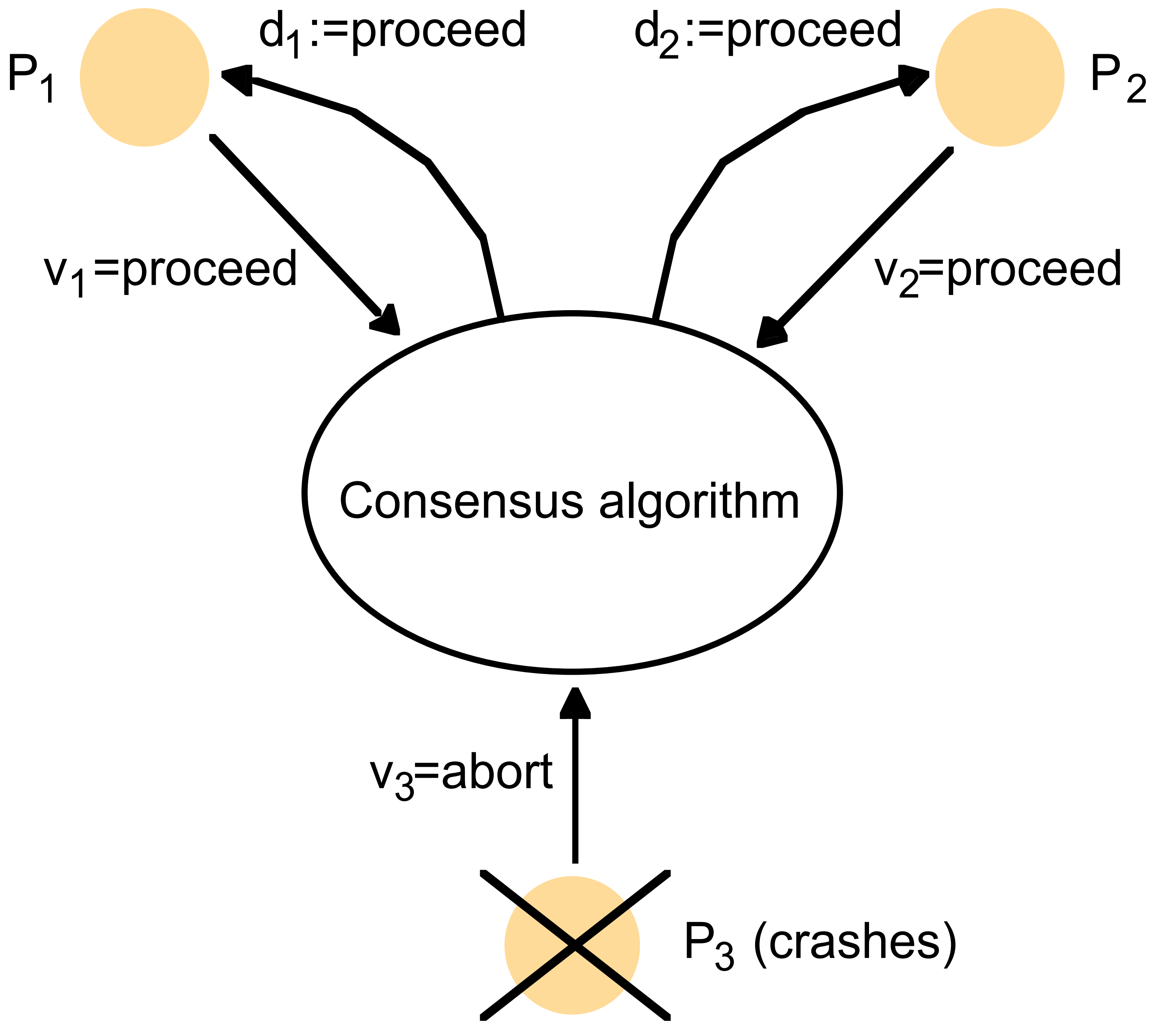
- Elect a leader
- Ring solution. Failure intolerant.
- Bully solution. Synchr., tolerates crash failures.

Consensus

The problem

- Each process p_i proposes a value from a set D
- Each process p_i eventually sets decision variable d_i
- The protocol must satisfy:
  Termination: each p_i sets its d_i
  Agreement: each correct p_i chooses the same d_i
  Integrity: If correct processes all proposed the same value, this value is the d_i.
- Failure model: Byzantine process failures, crash failures, reliable asynchronous communication.

Figure 15.16
Consensus for three processes



Variant: Byzantine Generals

- Only one process, the commander, proposes a value.
- *Agreement:* Correct processes must agree on some value.
- *Integrity:* If the commander is correct, they agree on his value.

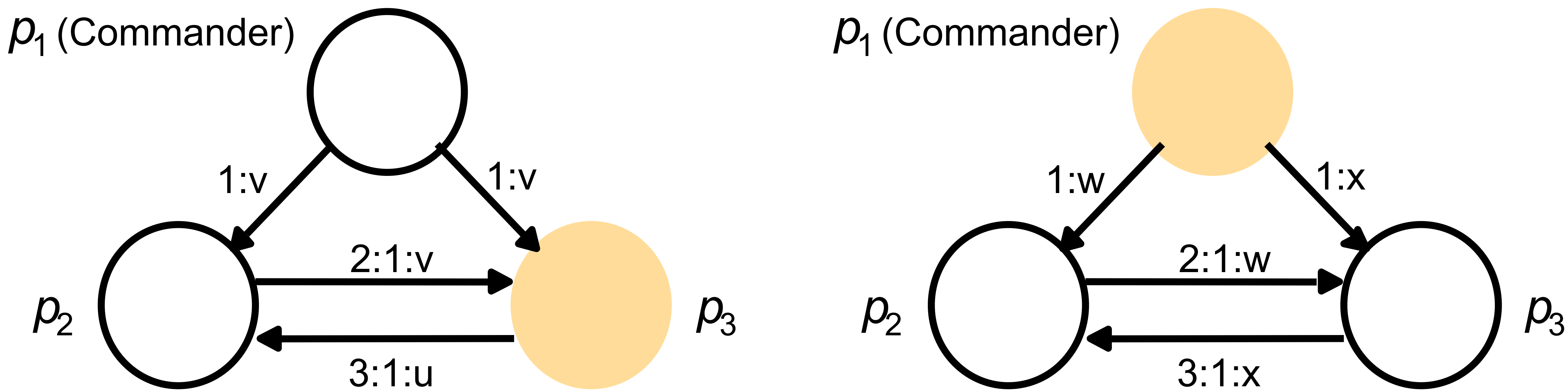
BG and C are equivalent

- If we had a protocol for Consensus, we could make one for Byzantine Generals:
    - Commander sends its proposed value to all.
    - Run C with each process proposing the value it received.
- If we had a protocol for Byzantine Generals, we could make one for Consensus:
    - Assume majority of processes correct.
    - Run BG once for each process to propose a value; then take majority.
    - (Proof in book using Interactive Consistency.)

BG in a synchronous setting

- Byzantine process failures, private communication.
- **No solution with 3 processes, 1 failure.**

Figure 15.18
Three Byzantine generals



Faulty processes are shown coloured

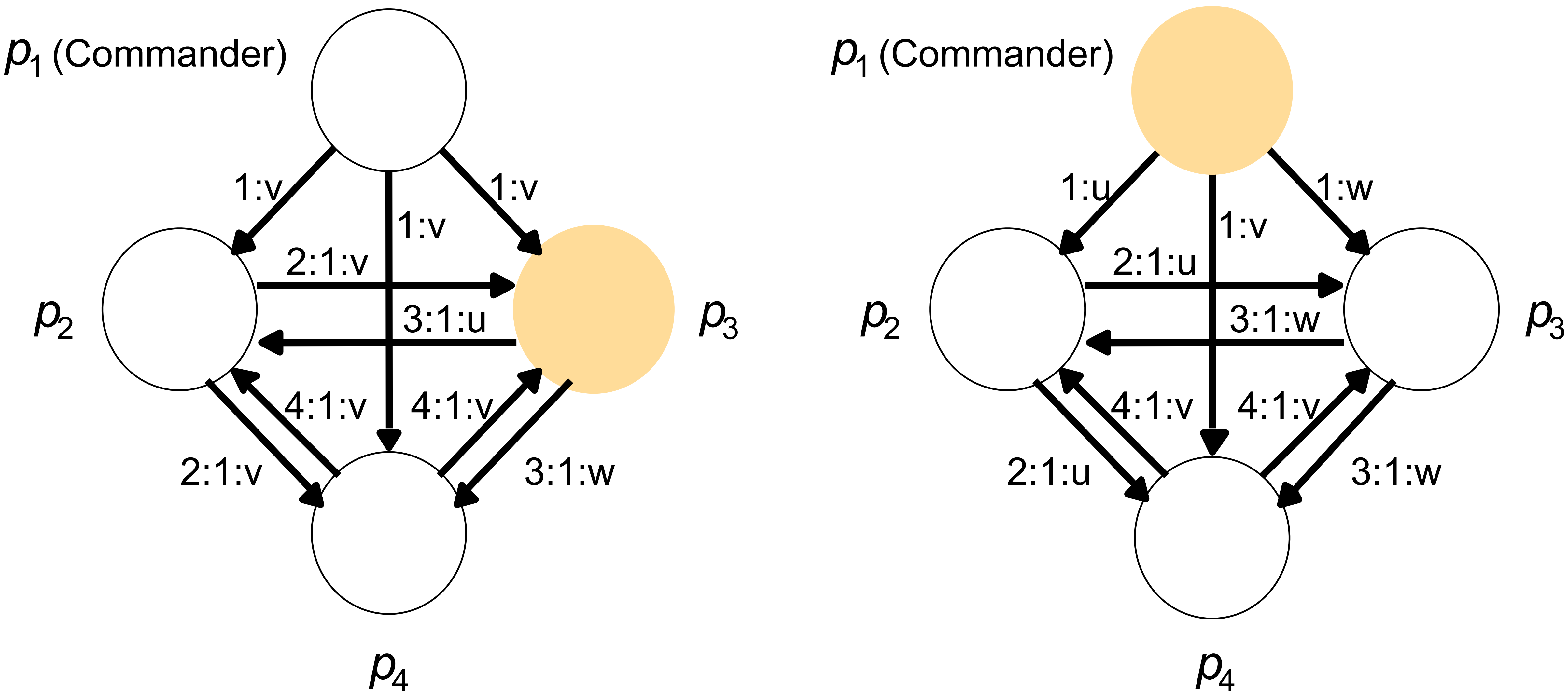
BG in a synchronous setting

- **No solution with 3 processes, 1 failure.**
- In, general no solution if $N \leq 3f$
  (f number of incorrect processes).
- If there were, we could have three processes simulate a larger system with N/3 generals each.
- If at most one process is faulty, it simulates at most N/3 faulty processes.
- This contradicts the previous example.

BG with N=4, f=1

- Special case of general algorithm.
- Two rounds:
  - Commander proposes value
  - Lieutenants exchange values

Figure 15.19
Four Byzantine generals



Faulty processes are shown coloured

Beyond the basics

- Impossible with just one crash failure in asynchronous systems. (Duh.)
- Practical workarounds:
  - Fault masking
  - Failure detectors
  - Randomized behaviour

Summary: Consensus

- Agree on a value.
- Byzantine Generals, Consensus. Equivalent.
- Impossible for 3 process, 1 byzantine process failure. In sync. system!
- In general possible for $N \leq 3f$ in sync system.
- Impossible with single process failure in async system.

Summary

Solution: Voting at the actuator

The engine receives commands from all computers, acts according to majority.

Summary

- Motivation: Fundamentals of agreeing.
- Distributed Mutual Exclusion
- Elections
- Consensus

Read on your own

- Maekawa’s voting algorithm.
- Interactive consistency
- Consensus protocol for a synchronous system
- Equivalence of BC, C
- Bully protocol details
- Impossibility and possibility results for BC/C.