IN.5022 — Concurrent and Distributed Computing

Coordination and Agreement

Prof. P. Felber

pascal.felber@unine.ch

Agenda

- Distributed mutual exclusion
  - Centralised, ring-based and distributed algorithms
- Leader election
  - Ring-based and distributed algorithms
- Consensus
  - Distributed algorithms

Distributed mutual exclusion (ME)

- Goal: synchronize concurrent accesses to shared resources
  - Printer, disk, file-locking service (UNIX `lockd`)
- Application-level protocol
  1. Enter critical section (block if necessary)
  2. Access resource
  3. Exit critical section (other processes may now enter)
- We assume that every process that is granted the resource eventually releases it

ME: specification

[ME1] (safety)

At most one process may be in the critical section at a time

[ME2] (liveness)

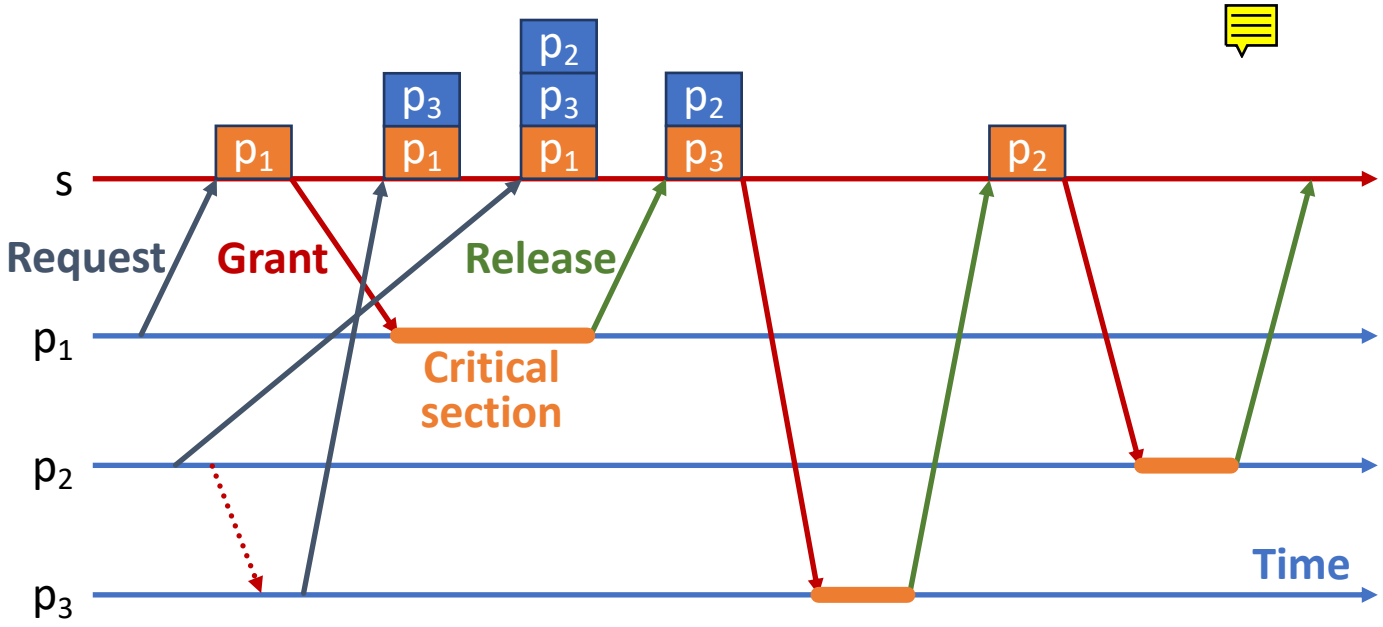
Requests to enter and exit the critical section eventually succeed (no starvation nor deadlock)

[ME3] (fairness/liveness) — optional

If one request to enter the critical section *happened-before* another, then entry is granted in that order

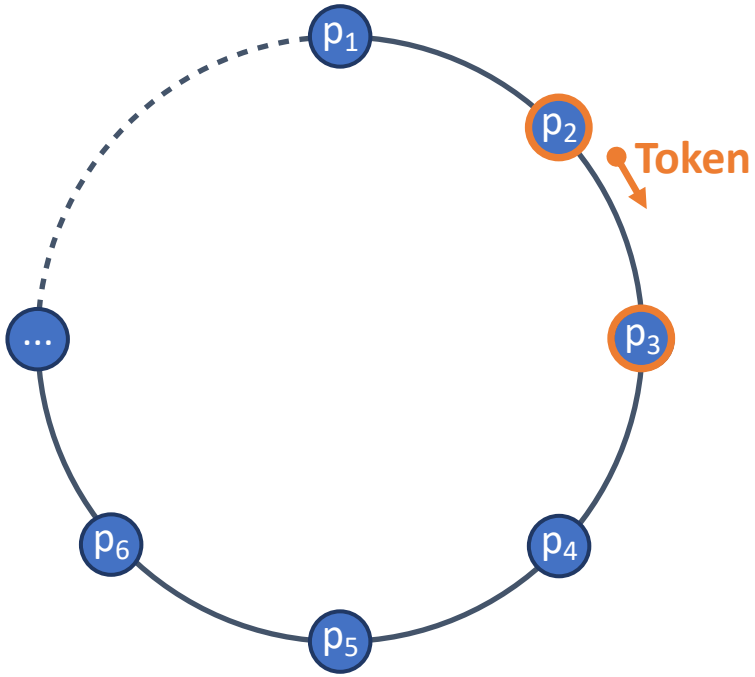
ME: centralized algorithm

- Server process allocates resources
  - FIFO queue of pending requests (first is granted resource)
  - Two messages to *enter*: request, grant
  - One message to *exit*: release
- Assumptions
  - Reliable channels
  - No failures
- Is ME3 satisfied?
  - Why?



ME: ring-based algorithm

- Token is passed around the ring
  - Process can enter CS when it receives token
  - Keeps token while in CS
  - On exit, pass token to next process (which may enter the critical section if desired)
- Assumptions
  - Reliable channels
  - No failures
- Is ME3 satisfied?
  - Why?



ME: distributed algorithm (Ricart–Agrawala)

- Process can enter CS after receiving go-ahead from all other processes
  - Assuming no failures, reliable FIFO channels

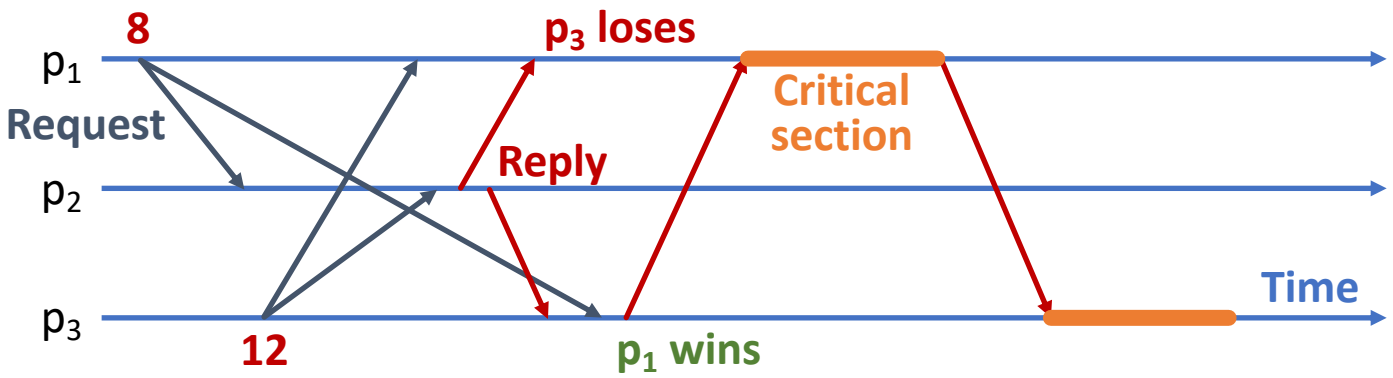
```
program RA-ME
define
  s: ∈ {RELEASED,      % CS state
        WANTED,
        HELD}
  t: integer      % logical clock
  Q: queue of requests % pending
initially s = RELEASED
```

```
CS-enter()              % enter critical section
  s := WANTED
  multicast(<REQUEST,t>) to all processes
  wait until received N-1 REPLY messages
  s := HELD
CS-exit()               % exit critical section
  s := RELEASED
  ∀<REQUEST,t_i,p_i> ∈ Q: send REPLY to p_i
  Q := []                                % empty Q

% To deliver request at p
do
» deliver(<REQUEST,t_i>) from p_i →
    if s = HELD or
        (s = WANTED and (t,p) < (t_i,p_i)) →
      Q := Q++<REQUEST,t_i,p_i>          % enqueue
    else
      send REPLY to p_i     % reply immediately
    fi
od
```

ME: distributed algorithm (cont'd)

- Is ME3 satisfied?
  - Why?



ME: distributed algorithm (cont'd)

- Proof of mutual exclusion
  - Assume that both p_i and p_j are in the CS concurrently
  - Therefore, both p_i and p_j have received N-1 replies
  - Therefore, p_i has granted access to p_j, and p_j to p_i
  - Assume, w.l.o.g., that p_i's request has a smaller timestamp than p_j's request
  - Therefore, p_i has sent its request before receiving p_j's request (due to the logical clock properties)
  - The algorithm prevents p_i from granting access to p_j if it has a pending request with a lower timestamp
  - Therefore, the algorithm implements mutual exclusion

ME: comparison of the algorithms

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems	Satisfies
Centralized	2 + 1	2	Coordinator crash	ME1, ME2
Token ring	1 to ∞	0 to N – 1	Lost token, process crash	ME1, ME2
Distributed	2·(N – 1) *(N with multicast)*	2 with multicast *(or non-blocking send)*	Crash of any process	ME1, ME2, ME3

Leader election (LE)

- Goal: choose a unique process to play a particular role (leader)
  - Any process can play the role, but only one has to
  - Example: leader allocates resources for mutual exclusion
- Any process can start (or call) an election
  - Several elections can be called concurrently, but the elected process must be unique
- Each process p_i has a variable $elected_i$ that contains the identifier of the leader or $\perp$
- Without loss of generality, the elected process is the non-failed process with the largest (unique) identifier
  - Identifier of p_i may be i, $<1/load_i, i>$, etc.

LE: specification

[LE1] (safety)

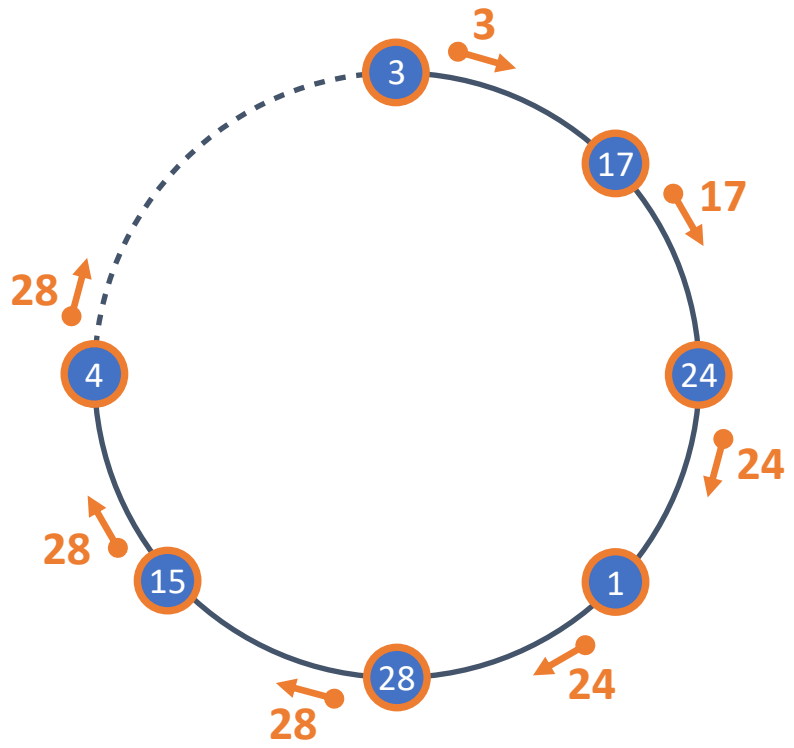
A participant process p_i has $elected_i = p$ or $elected_i = \perp$, where p is chosen as the non-crashed process at the end of the run with the largest identifier

[LE2] (liveness)

All processes p_i participate and eventually set $elected_i \neq \perp$, or they crash

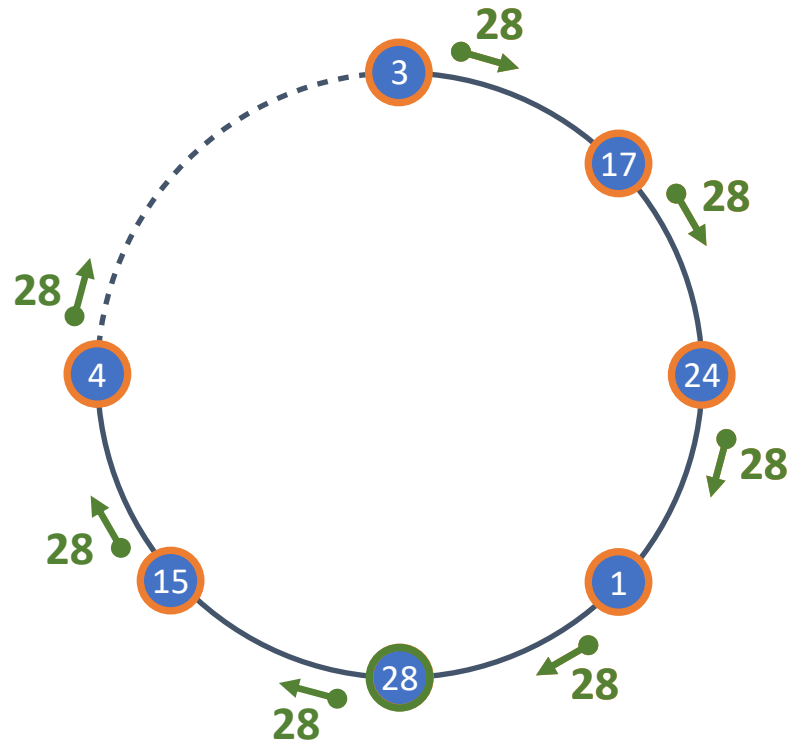
LE: ring-based algorithm

- Each process starts as non-participant
- Any process can start an election
  - Process becomes participant
  - Puts its identifier in an “election” message
  - Passes the message around the ring
- When non-participant receives message
  - Becomes participant
  - If message identifier is greater than its own, then forwards to next
  - If lower, then puts own identifier in message and forwards to next



LE: ring-based algorithm (cont'd)

- If process receives its own identifier
  - It must be the greatest ⇒ becomes leader
  - Sends an "elected" message with its own identifier along the ring (all processes become non-participant on the way)
- Leader discards elected message with its own identifier
- Assumptions
  - Reliable channels
  - No failures
- Nb. messages: 2N to 3N–1



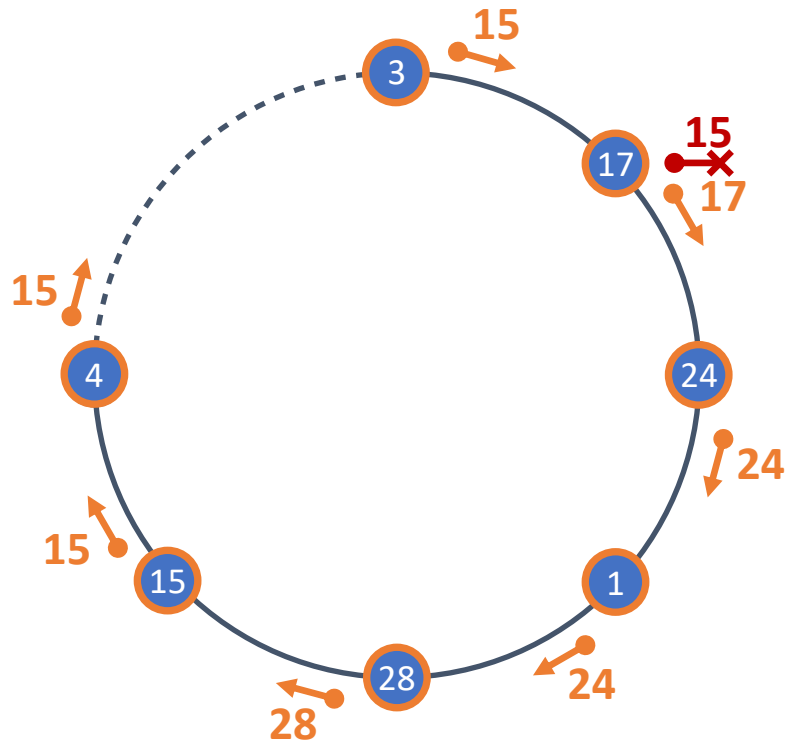
LE: ring-based algorithm (cont'd)

- Does the algorithm satisfy its specification?
- Safety
  - Only the greatest identifier traverses the whole ring (a process will not forward any lower identifier)
  - Only one process receives its own identifier back
  - No two processes can assign different values to $elected_i$
- Liveness
  - Follows from reliable links and no-failure assumption (guaranteed traversal of the ring)

LE: ring-based algorithm (cont'd)

- What if multiple processes call an election?
  - Participants do not forward election messages with identifier smaller than own
  - Only one election finishes
  - Competing election messages are extinguished as soon as possible

  ⇒ Role of the participant/non-participant status

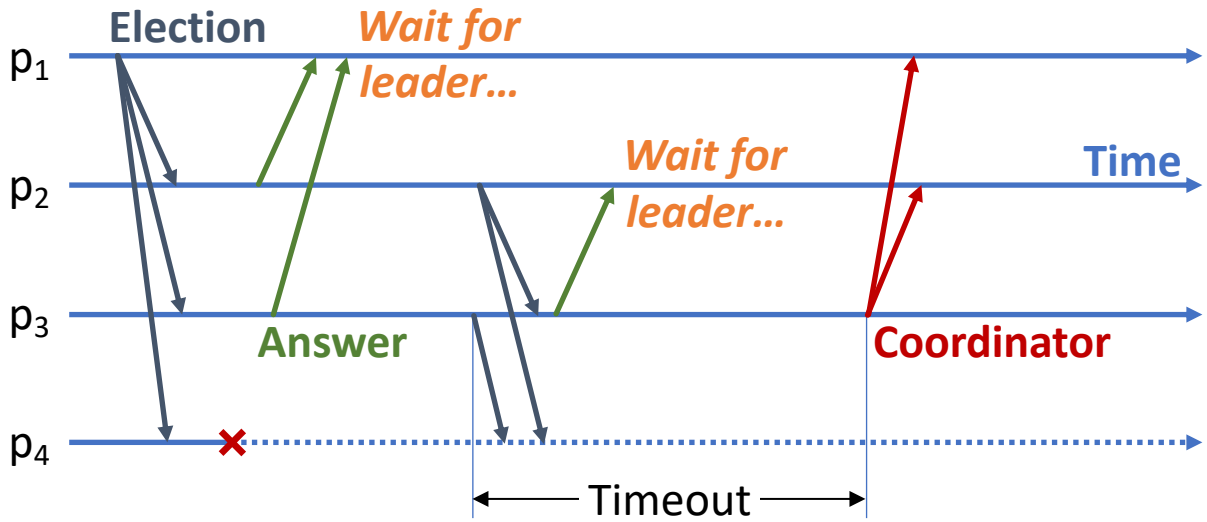


LE: “bully” algorithm

- Assumptions
  - Synchronous system
  - Processes know other processes and their identifiers a priori
    - Requires identifiers to be static (e.g., not dependent of the load)
    - Note that the process with the highest identifier can self-elect as leader
  - Processes can fail by crashing
    - If a crashed process recovers, it can call a new election
- “Bully” algorithm: the biggest guy in town always win

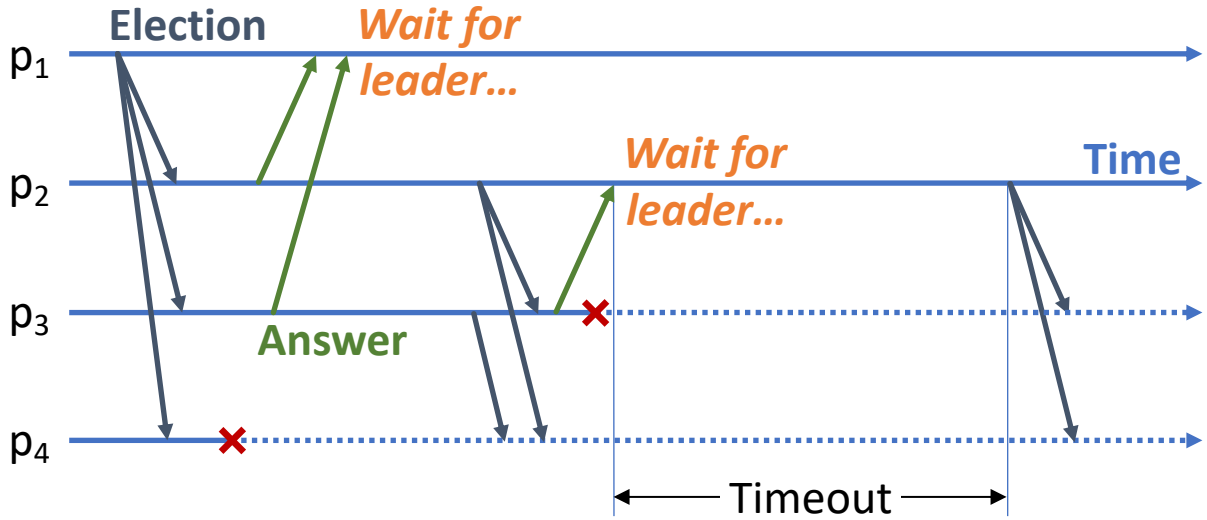
LE: “bully” algorithm (cont’d)

- To call an election
  - Process p sends “election” message to all processes with higher identifiers
  - If no one responds, p wins the election and becomes leader
  - If any answers, it takes over the election (i.e., starts an election if not already holding one) and p's job is done
- Once elected, leader sends a “coordinator” message to all



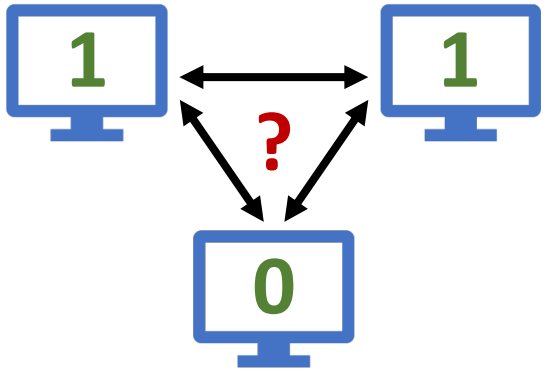
LE: “bully” algorithm (cont’d)

- What is a process fails after answering?
  - Waiting processes timeout and call a new election
- Safety is guaranteed if and only if processes are not replaced and timeouts are accurate (perfect FD)
- Liveness trivially follows from reliable channels
- Nb. messages: N–1 to $O(N^2)$



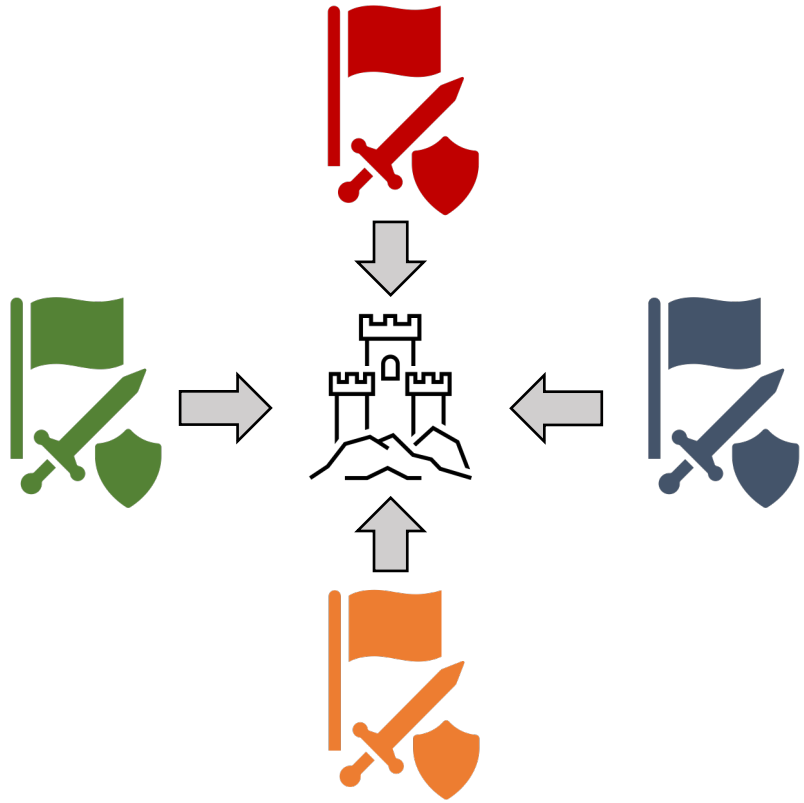
Distributed consensus

- In the consensus problem, the processes propose values and need to agree on *one among these values*
- Solving consensus is key to solving many practical problems in distributed computing
  - Total order broadcast, atomic commit, etc.



Consensus: example

- 4 allied armies, each led by a general, besiege a castle
  - An army without a general does not fight
  - To seize the castle, all four must attack
- Assumptions
  - Communications (routes, messengers) are *reliable*, but take unbounded time (asynchrony)
  - Generals may get killed (failures)
- Can generals reach consensus on “attack” or “withdraw”?



Consensus: specification

- A “symmetric” agreement problem
  - Processes start with (possibly divergent) opinions and must agree on one of them

Agreement: no two correct processes decide on different values

Validity: the value decided must have been the initial value of some process (non-triviality)

Termination: each correct process eventually decides on some value

Integrity: no process decides twice (a decision is irrevocable)

Uniform agreement — optional**:** no two processes (correct or not) decide on different values

Consensus: solvability

- Consensus cannot be solved (i.e., “guaranteed”) in an asynchronous systems [FLP85]
- (Uniform) consensus is solvable in a synchronous system
  - Both for crash failures and arbitrary failures
  - Can use timeouts to accurately detect failures
- Algorithm proceeds in a sequence of synchronous rounds
- Lower bound: to reach consensus despite up to f crash failures, an algorithm needs at least f+1 rounds of message exchanges

Consensus: algorithm for synchronous DS

- Algorithm assumes a synchronous system (bounded delays)
  - To simplify, each process proposes a different integer value
  - Tolerates f failures
  - Needs f+1 rounds
- Does the algorithm implement uniform consensus?

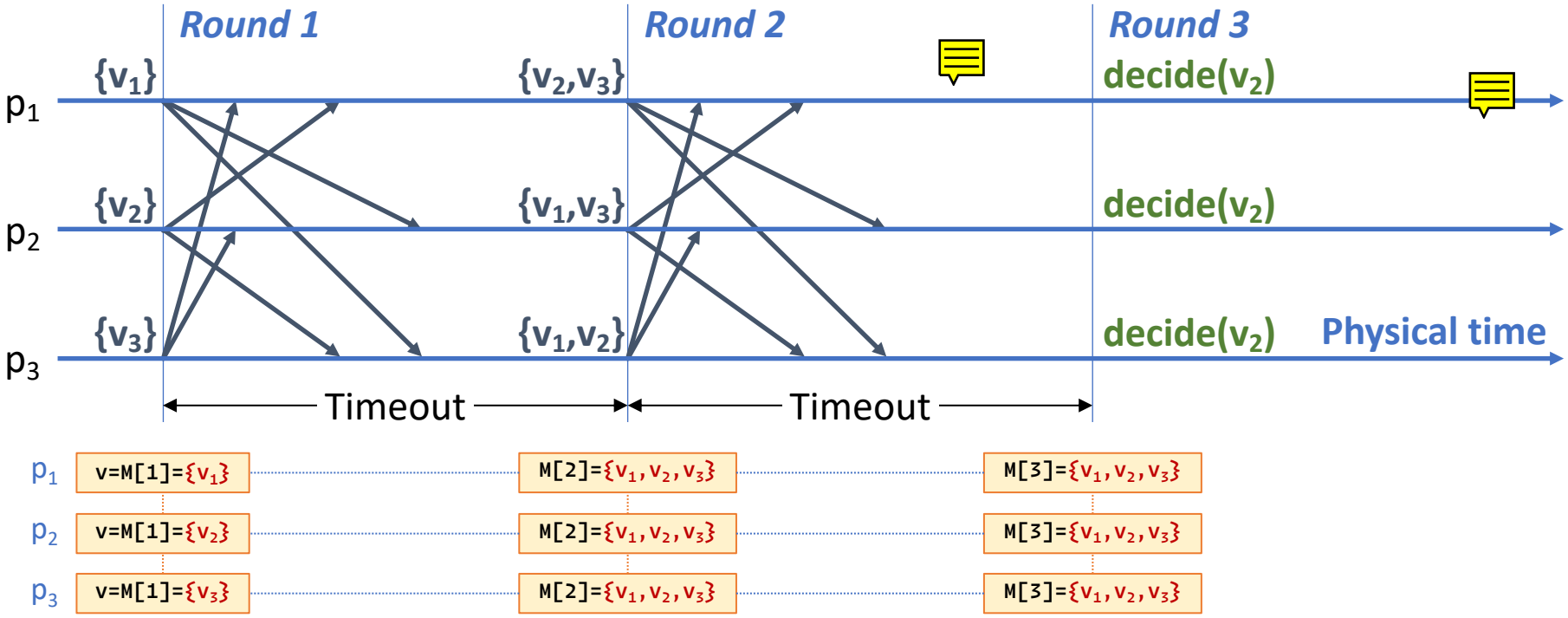
```
program Consensus
define
  r: integer                                   % round number
  v, d: integer  % value (proposal) and decision of p
  M: array of set of messages  % values in each round
initially r = 1, d = ⊥, M[0] = ∅, M[1] = {v}

% To run consensus with proposal v at p ∈ group G
while round r ≤ f+1 →                          % initial rounds
  B-multicast(G, M[r] \ M[r-1])     % new values only
  M[r+1] := M[r]
  do
  » B-deliver(Mj) from pj →
    M[r+1] := M[r+1] ∪ {Mj}
  » timeout →
    r := r + 1                              % move to next round
  od
% After f+1 round
d = min(M[f+1])              % same decision chosen by all
```

Consensus: sample run

Sample run with N=3, f=1, $v_2 < v_1 < v_3$

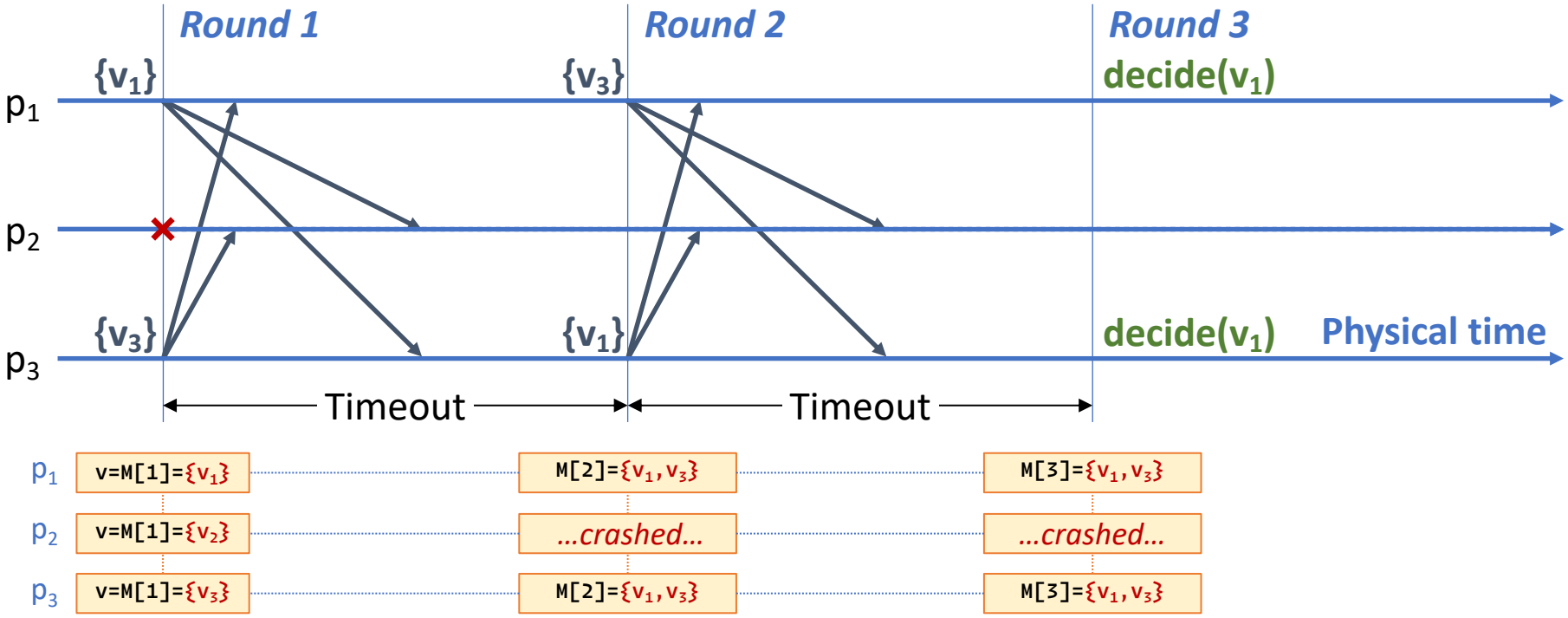
Do we *really* need 2 rounds?



Consensus: sample run (cont'd)

Sample run with N=3, f=1, $v_2 < v_1 < v_3$

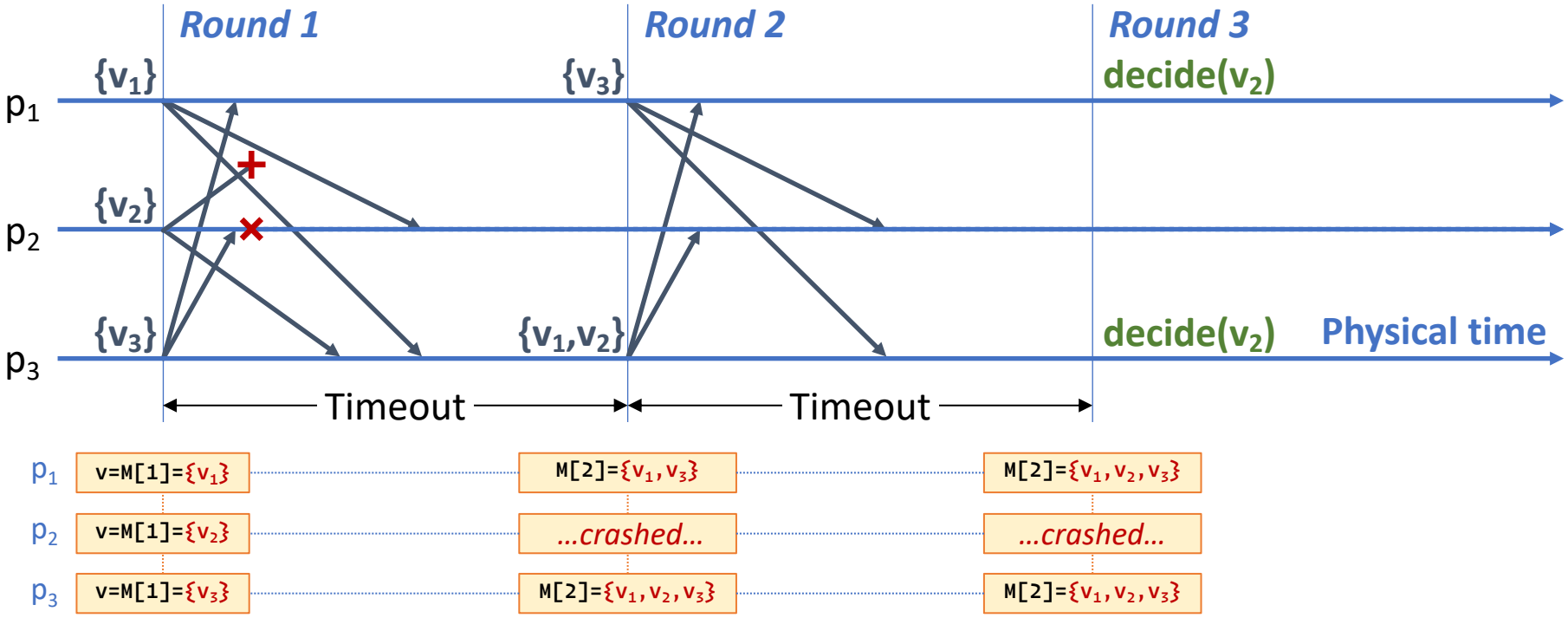
Do we *really* need 2 rounds, even upon failure (could decide in R2)?



Consensus: sample run (cont'd)

Sample run with N=3, f=1, $v_2 < v_1 < v_3$

Yes, we *do really* need 2 rounds (could not decide in R2)!



Practical consensus: Paxos [Lamport 89]

- A widely used consensus protocol proposed by Lamport
  - Production use by Google, Microsoft, Amazon, IBM, Apache, etc.
  - Mainly used for state machine replication (SMR): agree on ordered set operations (with associated) input to apply by each replica
  - Three roles: *proposer*, *acceptor* and *learner*
  - Two phases: *prepare* and *accept*
- Operates in a sequence of rounds to deal with failures
  - Each view chooses a value and then seeks a decision "quorum"
  - A later view chooses any possible earlier decision
  - Paxos is *not* guaranteed to terminate (FLP impossibility)

"Paxos made simple" [L. Lamport] : `https://lamport.azurewebsites.net/pubs/paxos-simple.pdf`

Raft protocol *(an alternative)* : `https://raft.github.io/ – https://github.com/rabbitmq/ra`

Summary


- Coordination and agreement are at the core of many problems in distributed systems
  - Mutual exclusion: ensure that only one process can execute at a time (synchronize concurrent accesses to shared resources)
  - Leader election: select a unique distinguished process to play a particular role
  - Consensus: let multiple processes agree on a common decision
    - Can be the leader for LE or the process entering a critical section for ME!
- Algorithms can be implemented in (partially) synchronous DS with some faulty processes
  - Need additional assumptions or weaker properties in asynchronous DS