Midterm Exam 06: Exercise 2
Atomic Shared Memory

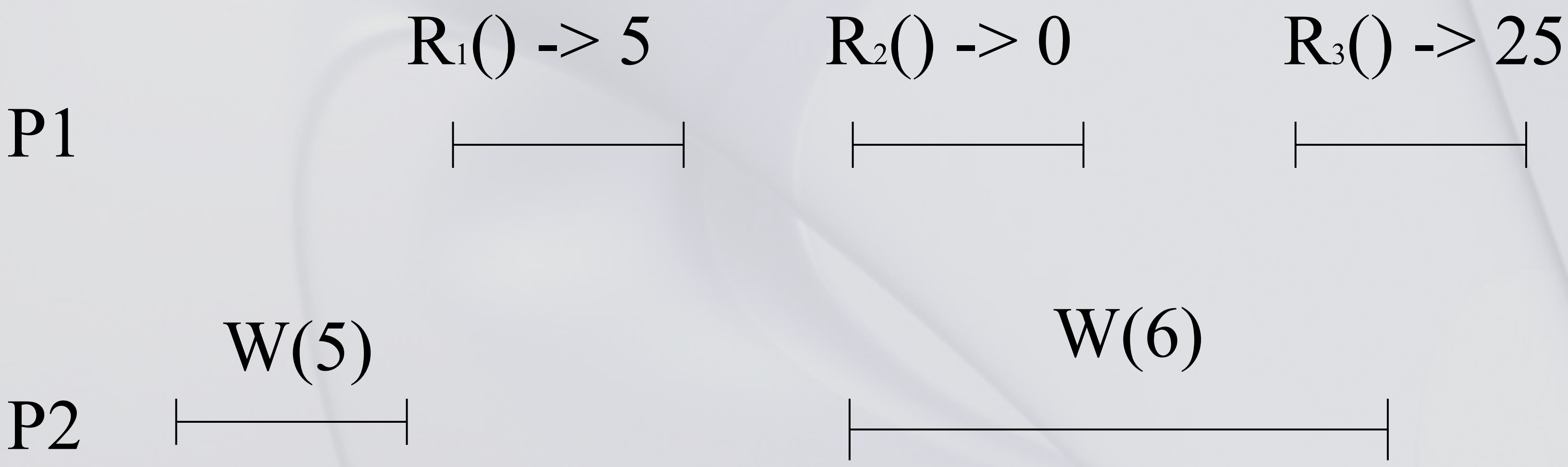
Atomic register

- **Every failed (write) operation appears to be either complete or not to have been invoked at all**

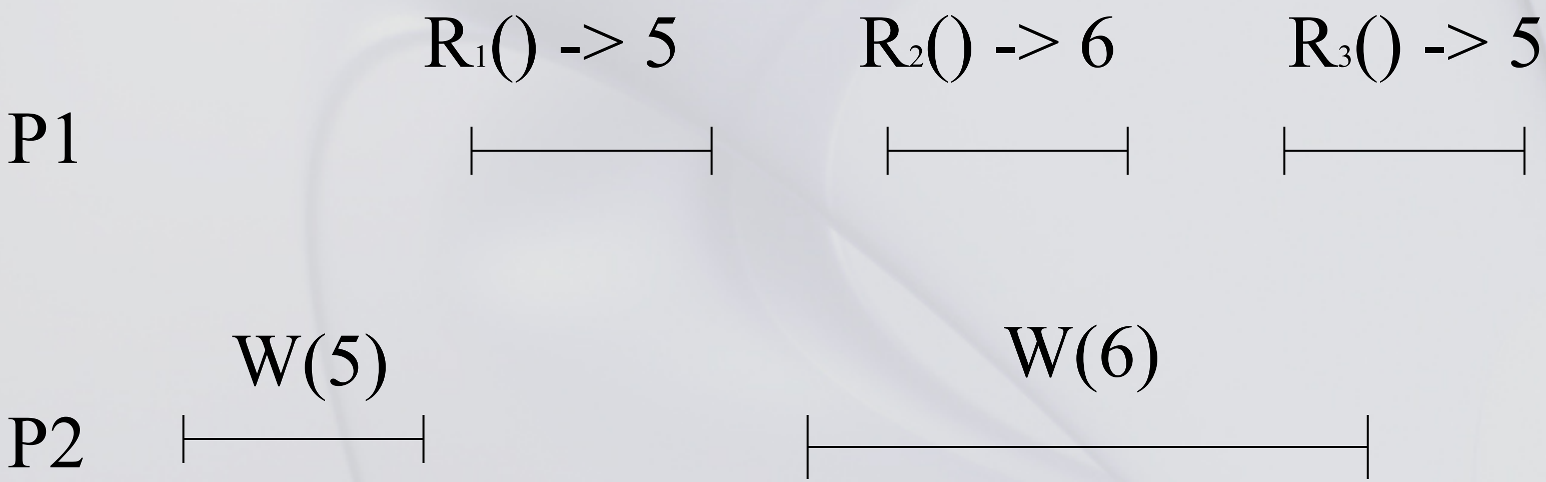
And

- **Every complete operation appears to be executed at some instant between its invocation and reply time events**
- **In other words, atomic register is:**
  - Regular (READ returns the latest value written, or one of the values written concurrently), **and**
  - READ rd' that follows some (complete) read rd does not return an older value (than rd)

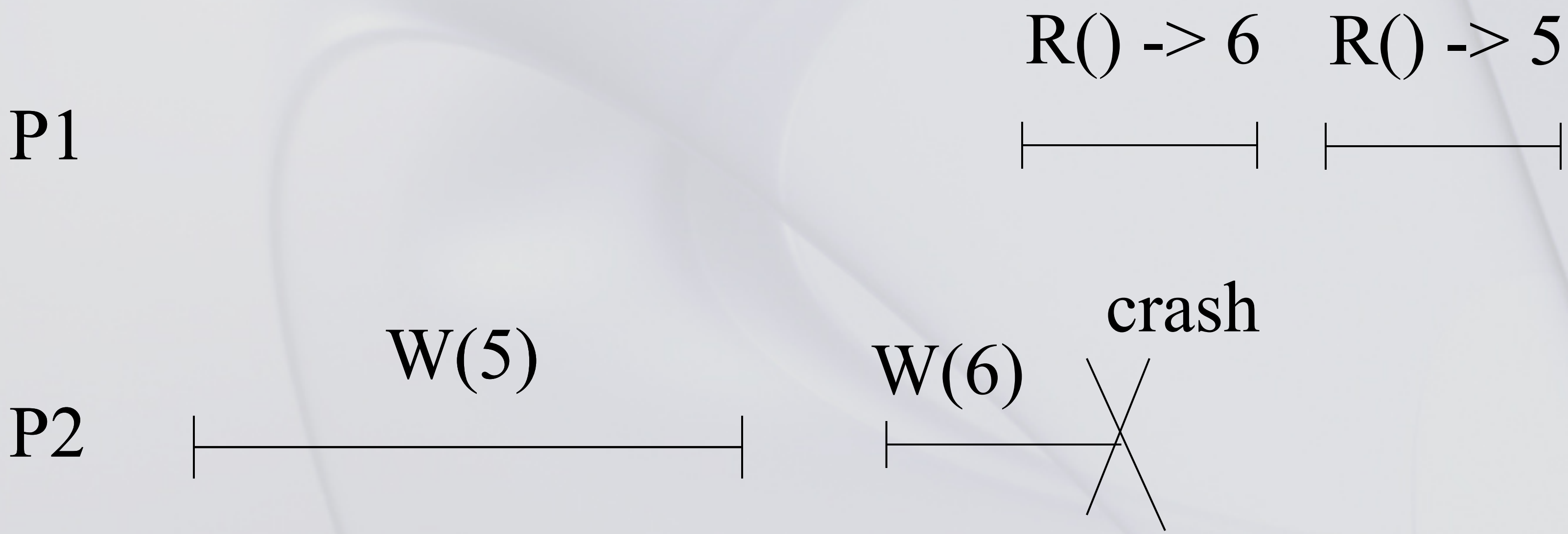
Non-Atomic Execution 1



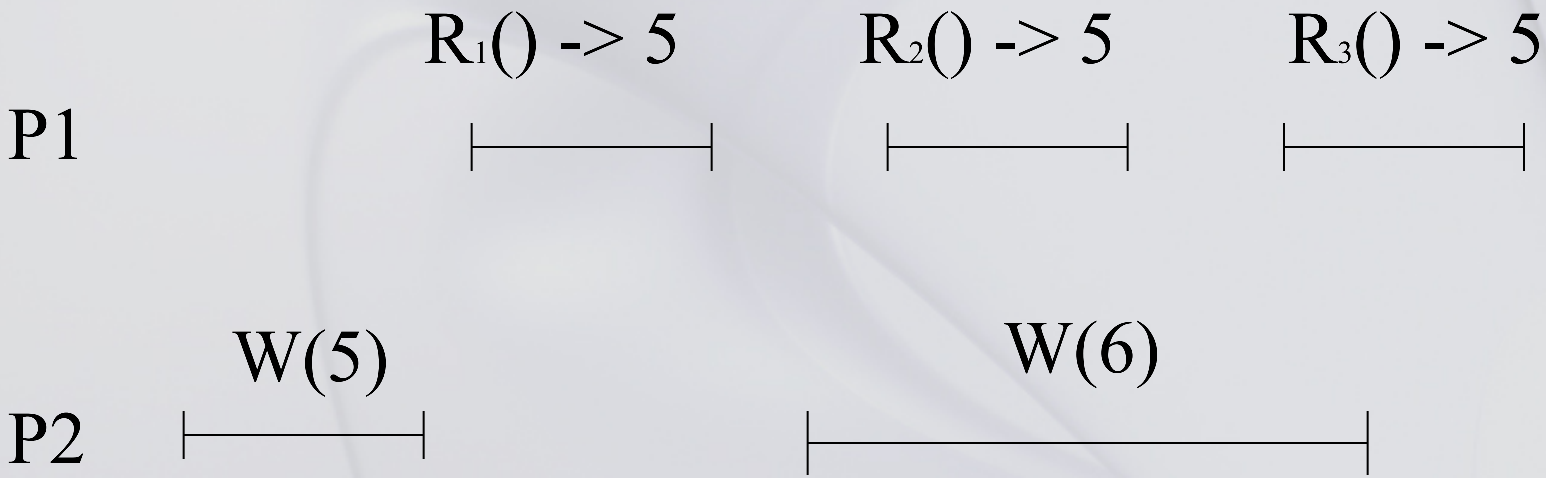
Non-Atomic Execution 2



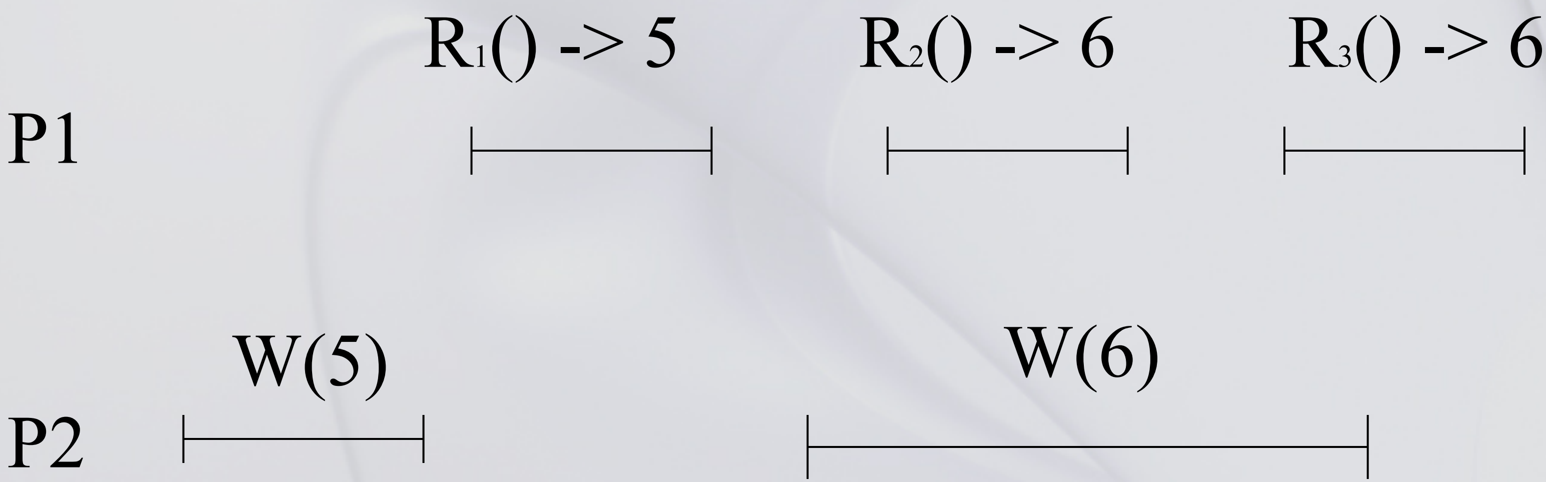
Non-Atomic Execution 3



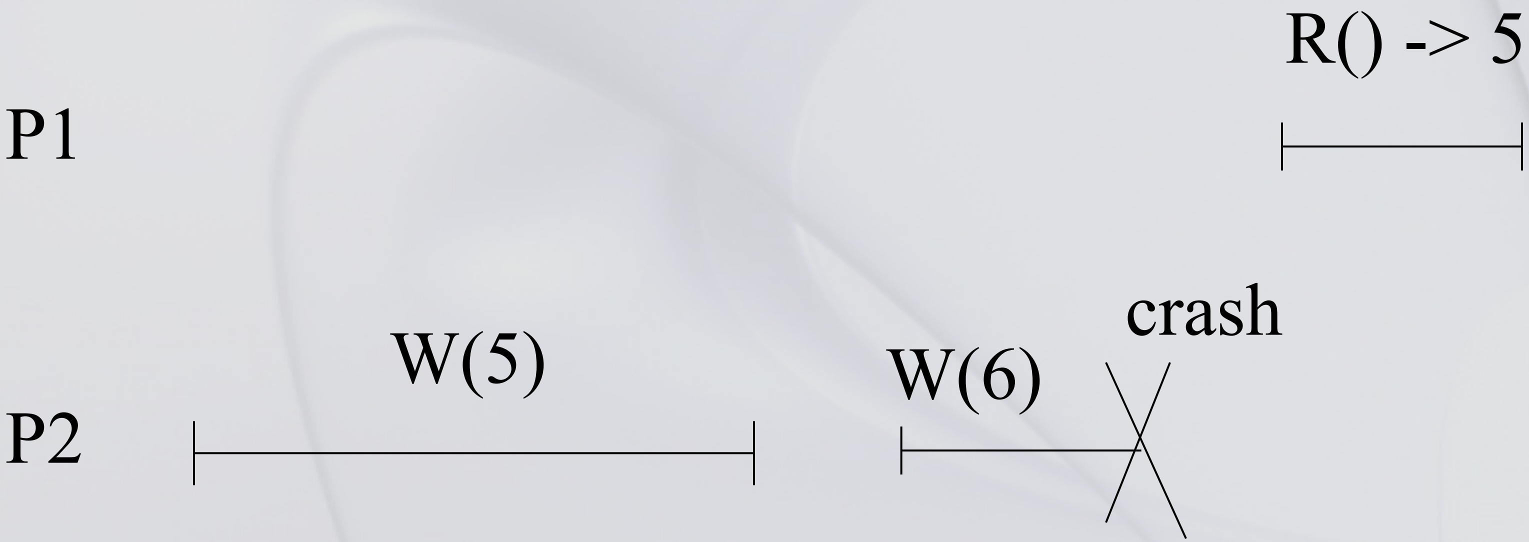
Atomic Execution 1



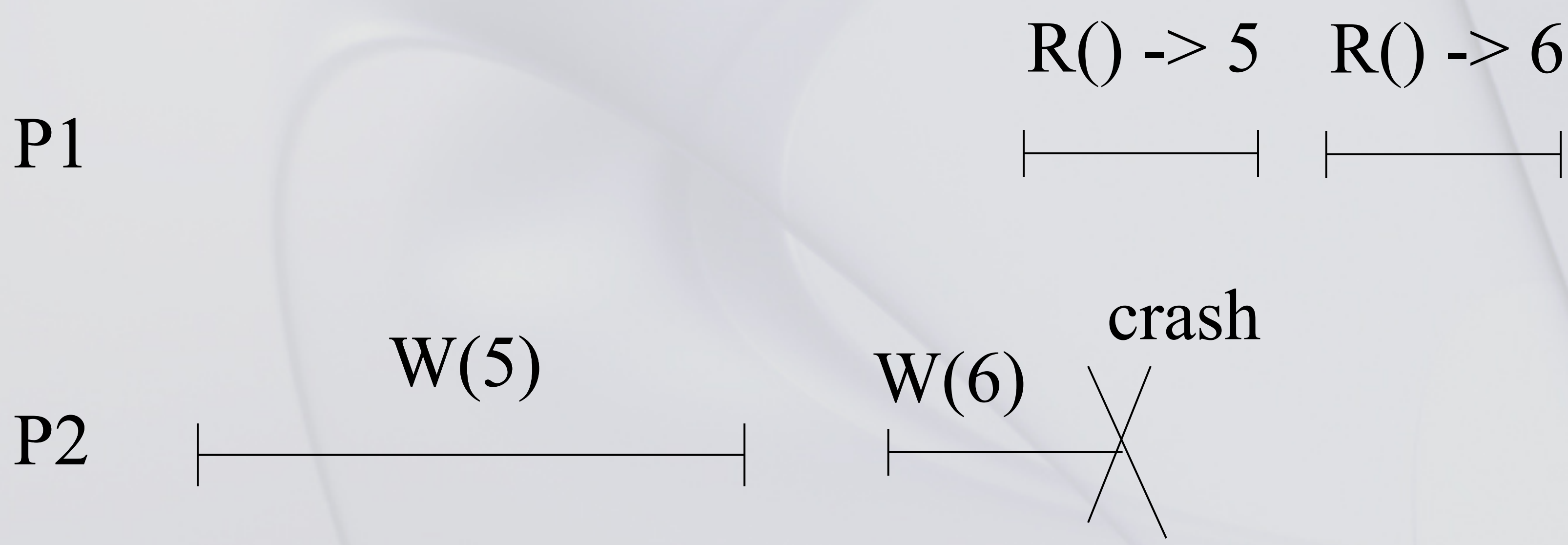
Atomic Execution 2



Atomic Execution 3



Atomic Execution 4



Best case complexity

- **We build algorithms for the worst case (or *unlucky*) situations**
  - Asynchrony
  - Concurrency
  - Many failures
- **However, very frequently situation is not that bad (*lucky* executions)**
  - Synchrony
  - No concurrency
  - Few failures (or none at all)
- **Practical algorithms should take advantage of the *lucky* executions**

Exercise 2

- **Give a 1-writer n-reader atomic register implementation (n=5, majority of processes is correct) in which**
  - L2: all read/write operations* complete in at most 2 round-trips
    - In every round-trip, a client (writer or reader) sends a message to all processes and awaits response from some subset of processes
  - L1: all *lucky* read/write operations* should complete in a single round-trip
    - A read/write operation *op* is *lucky* if:
      - The system is synchronous: messages among correct processes delivered within the time Δ (known to all correct processes)
      - *op* is not concurrent with any write operation
      - At most one process is faulty

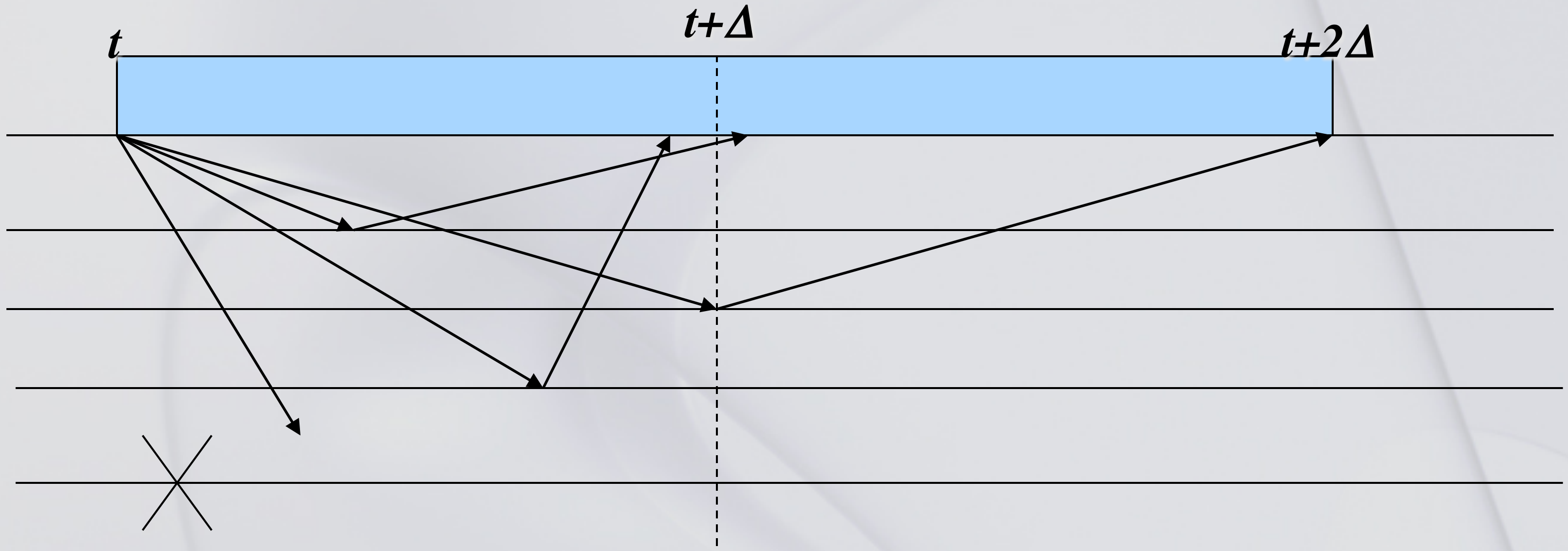
**invoked by a correct client*

The majority algorithm [ABD95]

- **All reads and writes complete in a single round-trip**
  - A client sends a message to all processes and waits for response from a majority
- **However, this algorithm implements only *regular* register (not *atomic*)**
- **To make the algorithm atomic:**
  - readers impose a value with a highest timestamp to a majority of processes
    (requires a second round-trip)

Lucky operations

- If the operation is lucky, the client will be able to receive (at least) 4 (out of 5) responses



timer=2Δ

Solution

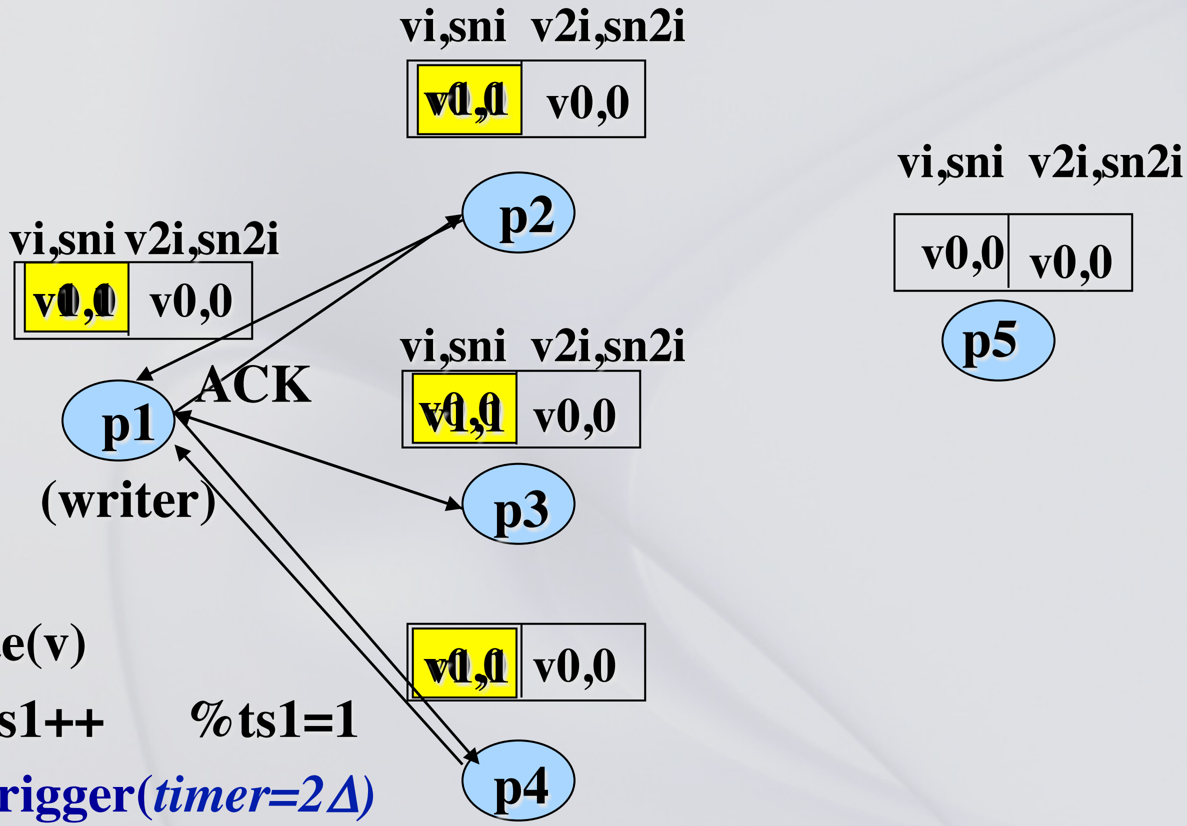
- **In the following slides we modify the majority algorithm of [ABD95]**
  - [ABD95] is given in slides 30-32, Regular Register Algorithms lecture notes – in Shared Memory part 2

Algorithm - Write()

- **Write(v) at p1 (the writer)**
  - ts1++
  - **trigger(*timer=2Δ*)**
  - send [W,ts1,v] to all
  - when receive [W,ts1,ack] from majority
  - **Wait for** expiration of *timer*
  - **If** *received 4 acks* **then**
    - Return ok
  - **else**
    - Send [W2,ts1,v] to all
    - when receive [W2,ts1,ack] from majority
    - Return ok

- **At pi**
  - when receive [W,ts1, v] from p1
  - If ts1 > sni then
    - vi := v
    - sni := ts1
  - send [W,ts1,ack] to p1
  - when receive [W2,ts1, v] from p1
  - If ts1 > sni2 then
    - vi2:= v
    - sni2 := ts1
  - send [W2,ts1,ack] to p1

How the (*lucky*) write works



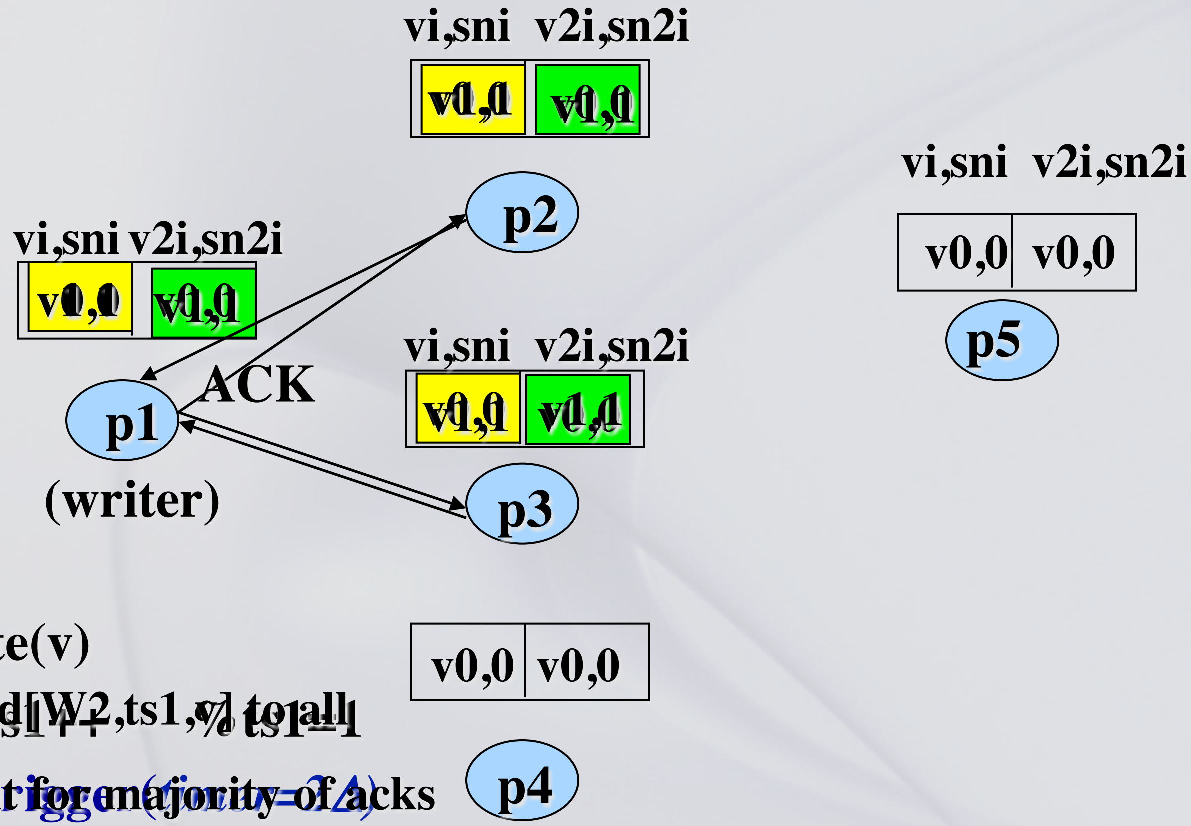
write(v)

ts1++ **%ts1=1**

trigger(*timer=2Δ*)

send [W,ts1,v] to all

How the *(unlucky)* write works



write(v)

send [W2,ts1,v] to all

ts1++ % ts1=1

wait for majority of acks

trigger(done=2Δ)

send [W,ts1,v] to all

Algorithm - Read()

- **Read() at pi**
  - rsi++
  - **trigger(*timer=2Δ)***
  - send [R,rsi] to all
  - when receive [R,rsi,snj,vj] from majority
  - v := vj with the largest snj
    - Return v
- **At pi**
  - when receive [R,rsj] from pj
  - send [R,rsj,sni,vi] to pj

Algorithm - Read()

- **Read() at pi**
  - rsi++
  - **trigger(*timer=2Δ*)**
  - send [R,rsi] to all
  - when receive [R,rsi,snj,vj,sn2j,v2j] from majority
  - **Wait for** expiration of *timer*
  - v := vj with the largest snj
    - Return v
- **At pi**
  - when receive [R,rsj] from pj
  - send [R,rsj,sni,vi,sn2i,v2i] to pj

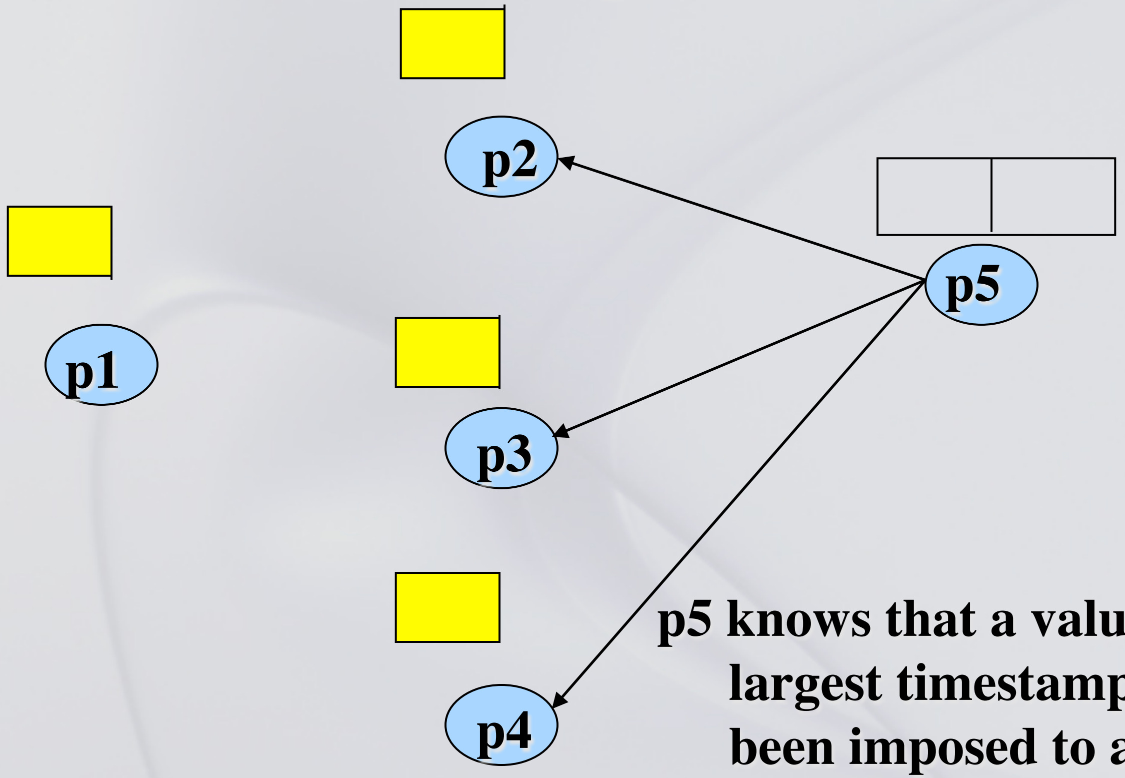
Algorithm - Read()

- **Read() at pi**
  - rsi++
  - **trigger(*timer=2Δ*)**
  - send [R,rsi] to all
  - when receive [R,rsi,snj,vj,sn2j,v2j] from majority
  - **Wait for** expiration of *timer*
  - v := vj or v2j with the largest snj or sn2j
  - **If** v is some v2j **or** there are 3 responses where vj=v and snj=snMAX **then**
    - Return v
  - **else**
    - Send [W,ts1,v] to all
    - when receive [W,ts1,ack] from majority
    - Return ok

- **At pi**
  - when receive [R,rsj] from pj
  - send [R,rsj,sni,vi,sn2i,v2i] to pj

How the *lucky* read works

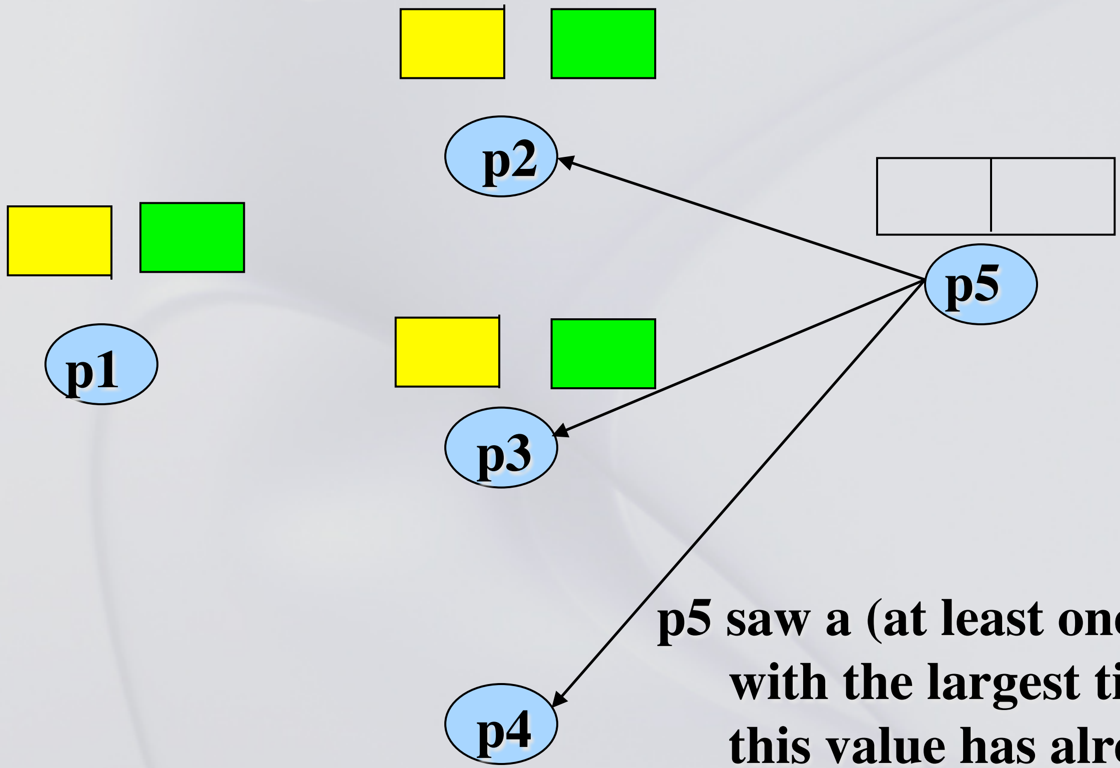
Following the *lucky* write



p5 knows that a value with the largest timestamp has already been imposed to a majority (in our case p2,p3 and p4)

How the *lucky* read works

Following the *unlucky (2 round-trip)* write



p5 saw a (at least one) « green » value with the largest timestamp: hence, this value has already been imposed to a majority in the 1st round-trip of the write

Unlucky read

- **Must impose a value with the largest timestamp to a majority of processes**
  - **If** v is some v2j **or** there are 3 responses where vj=v and snj=snMAX **then**
    - Return v
  - **else**
    - Send [W,ts1,v] to all
    - when receive [W,ts1,ack] from majority
    - Return ok
- **W not W2!**
  - Readers impose a value on « yellow » not « green » variables
  - Only the writer writes into the « green » variables (v2i,sn2i)

An offline exercise

- Try to rigorously prove correctness of this algorithm
- Proving correctness may appear on the final exam