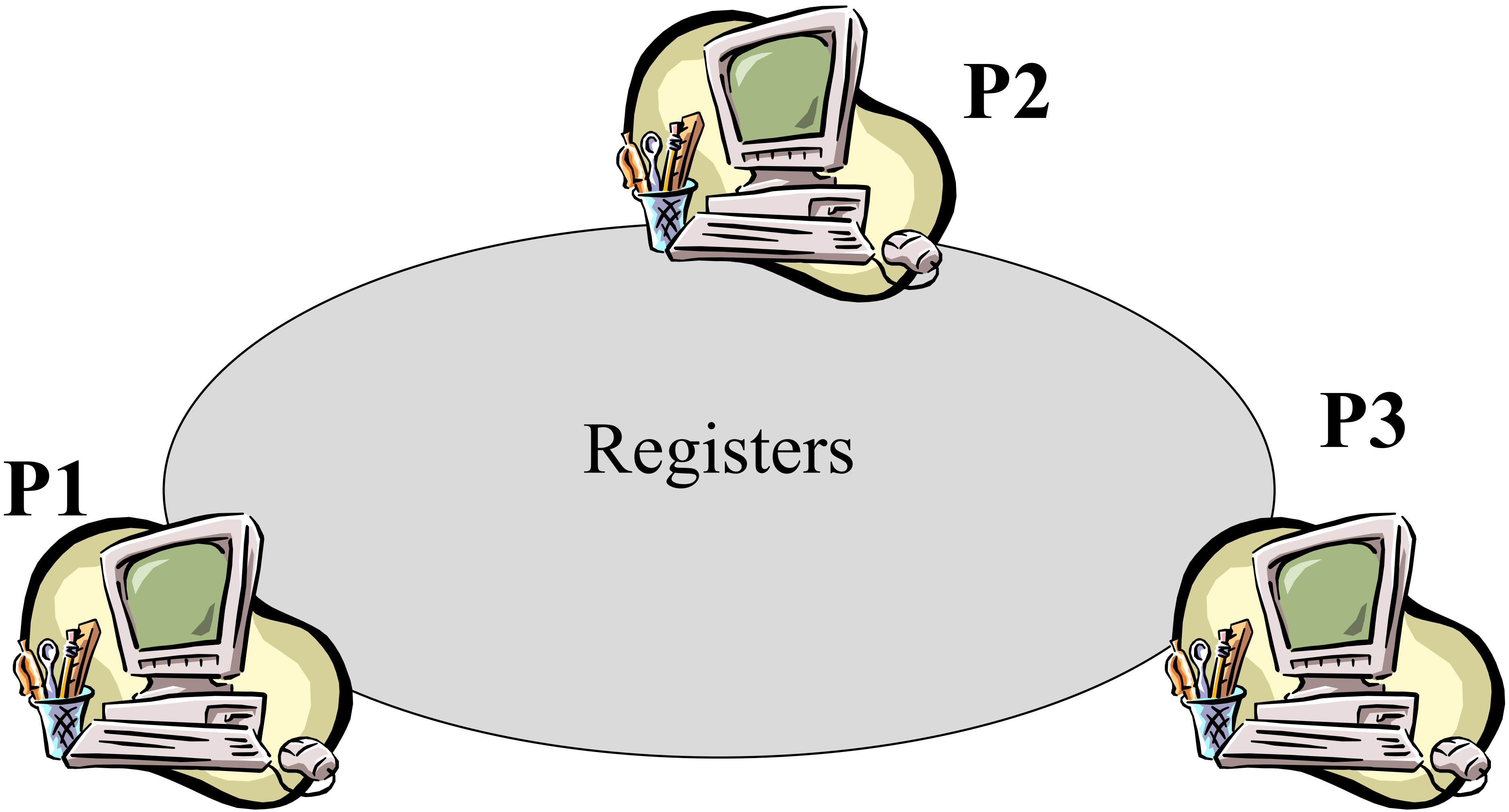
From Message Passing to Shared Memory

R. Guerraoui
Distributed Computing Laboratory
lcdwww.epfl.ch

The goal



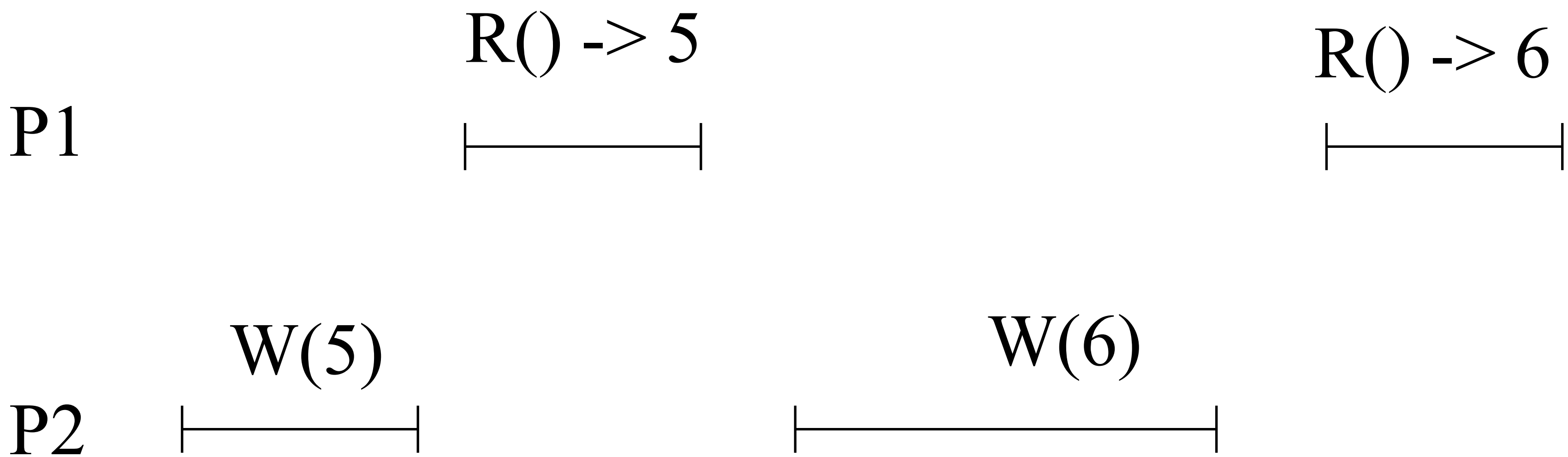
Register: Specification

- A register contains ***integers :*** initial value 0

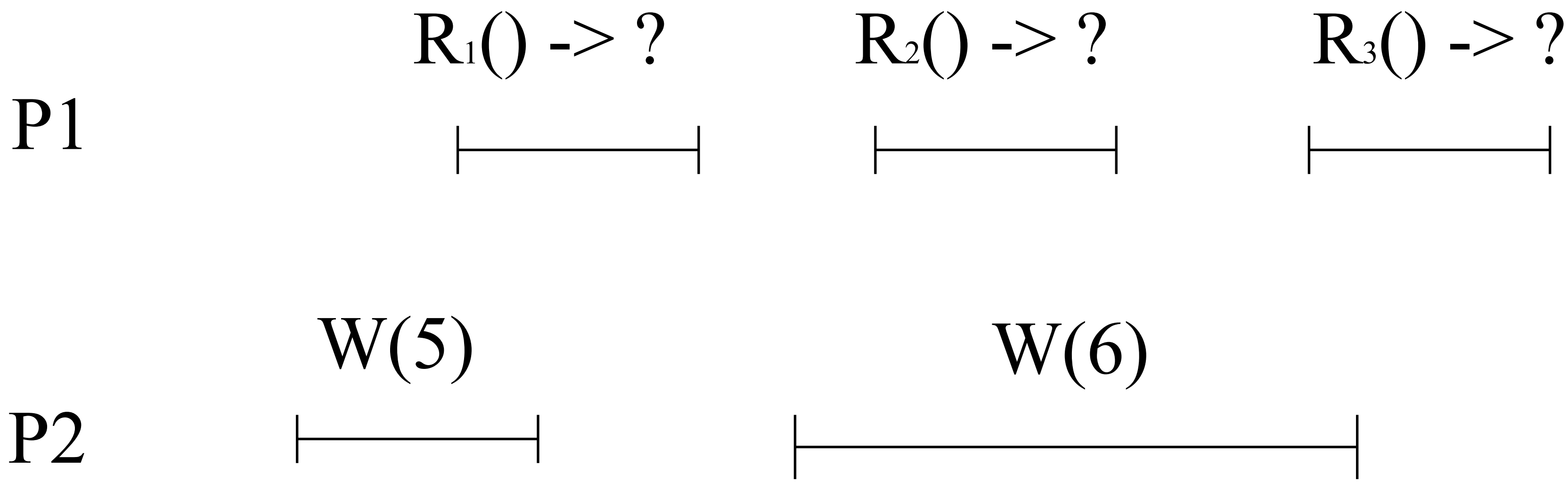
- Every value written is ***uniquely*** identified (this can be ensured by associating a process id and a timestamp with the value)

- Assume a register is local to a process, i.e., accessed only by one process: the value returned by a ***Read()*** is the last value written

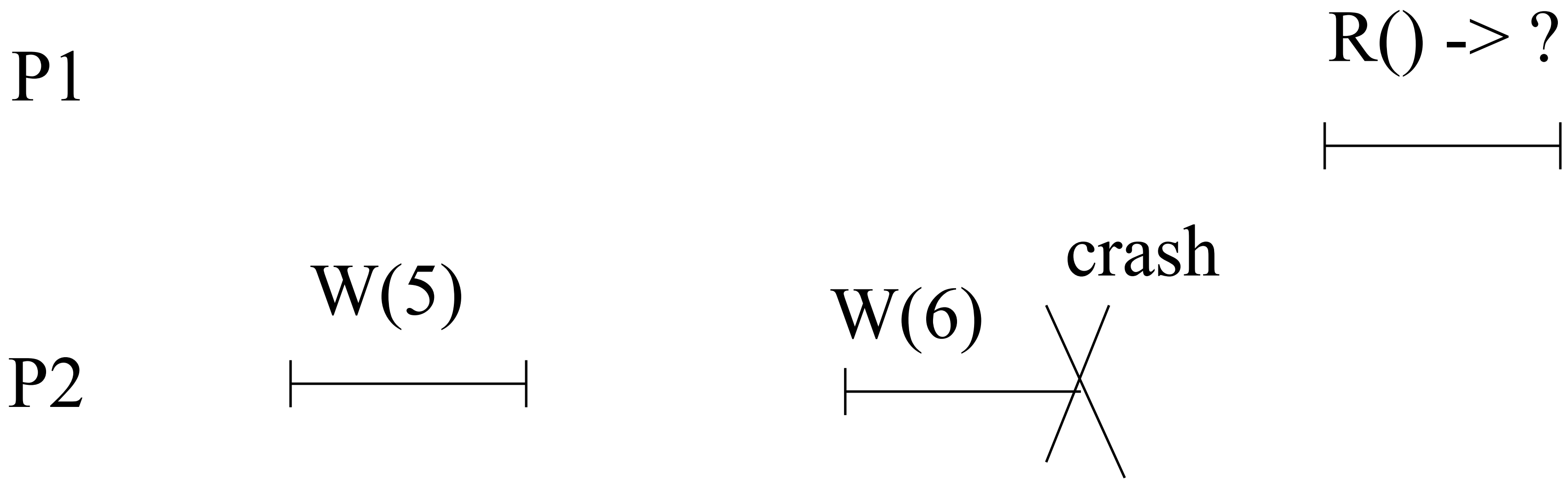
Sequential execution



Concurrent execution



Execution with failures



Regular register

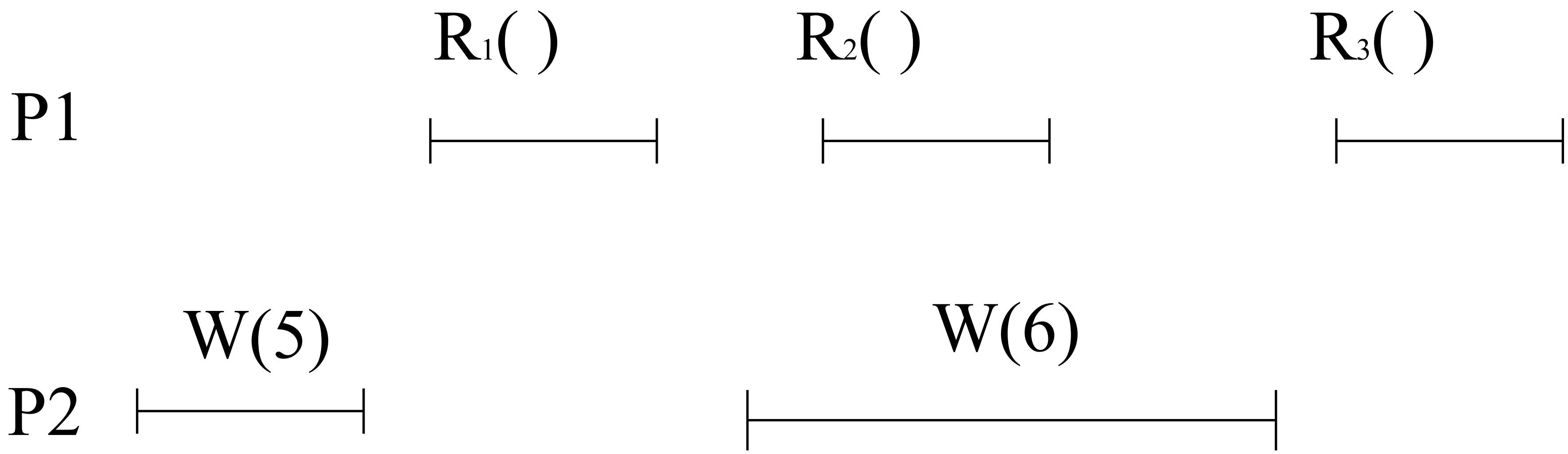
- Assumes only ***one*** writer

- Provides ***strong*** guarantees when there is no concurrent operations

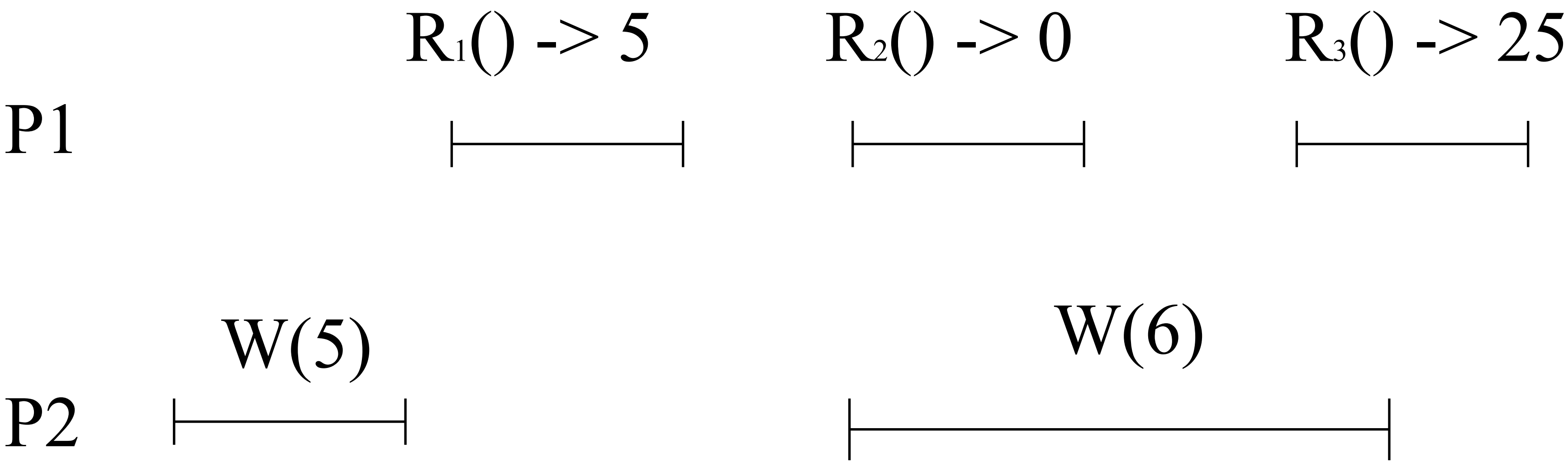
- When some operations are concurrent, the register provides ***minimal*** guarantees

- ***Read()*** returns:
  - ✓***the last value*** written if there is no concurrent or failed operations
  - ✓otherwise the last value written or ***any*** value concurrently written, i.e., the input parameter of some ***Write()***

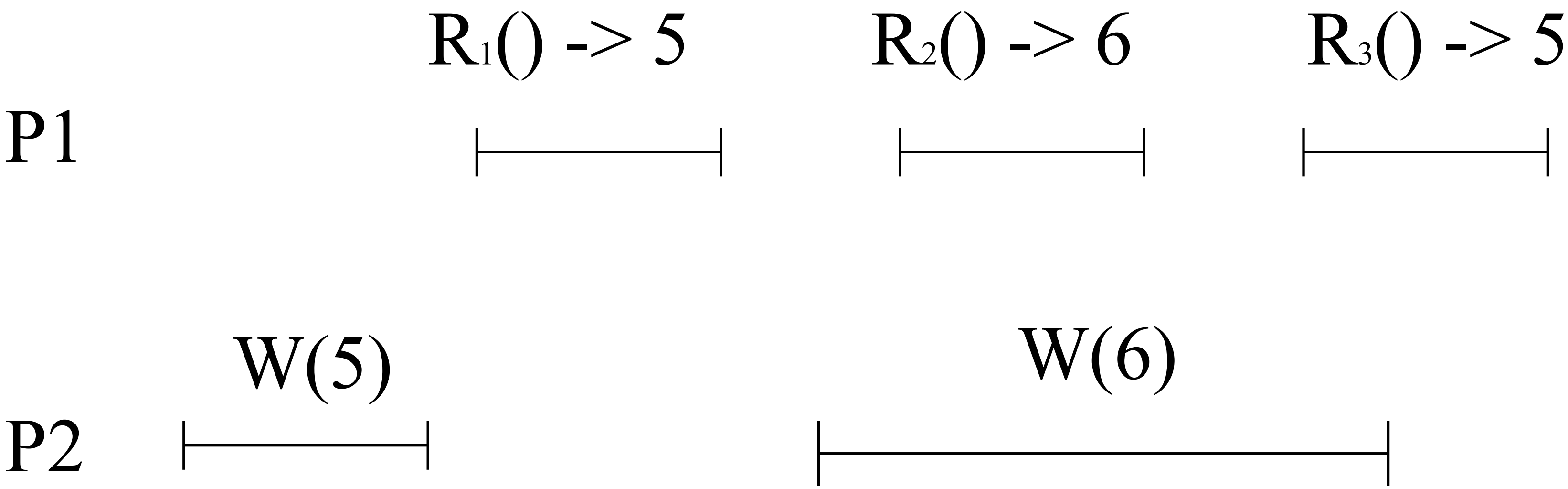
Execution



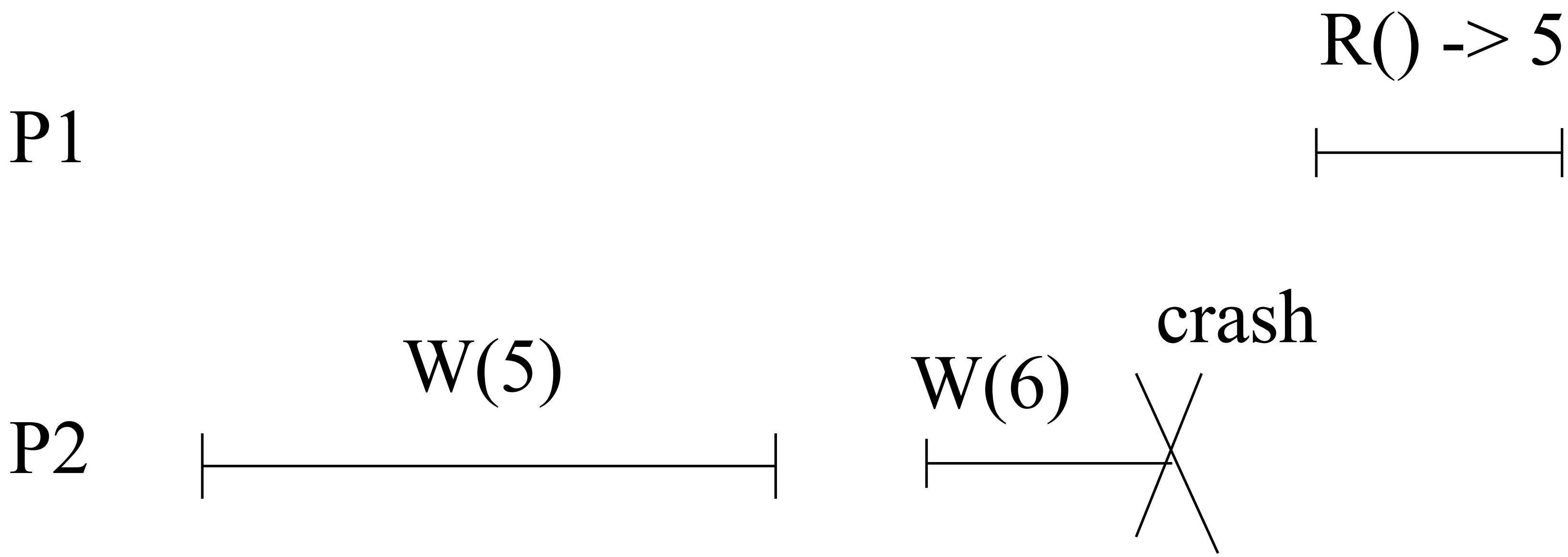
Results 1



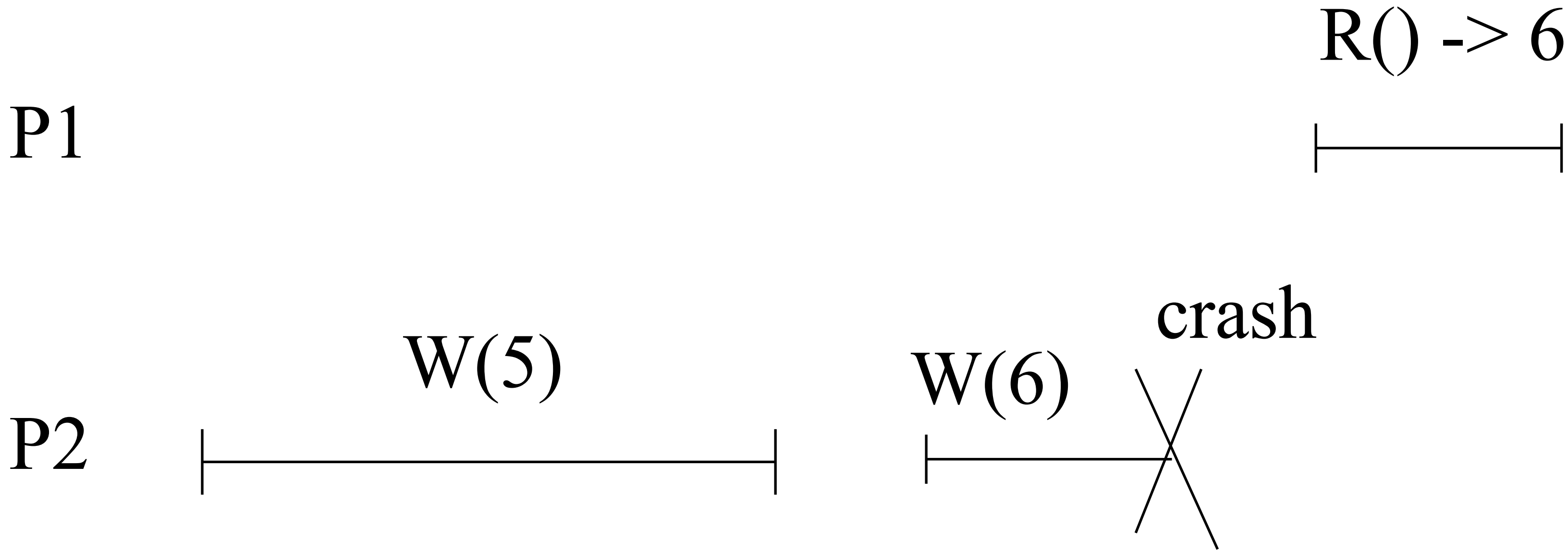
Results 2



Results 3



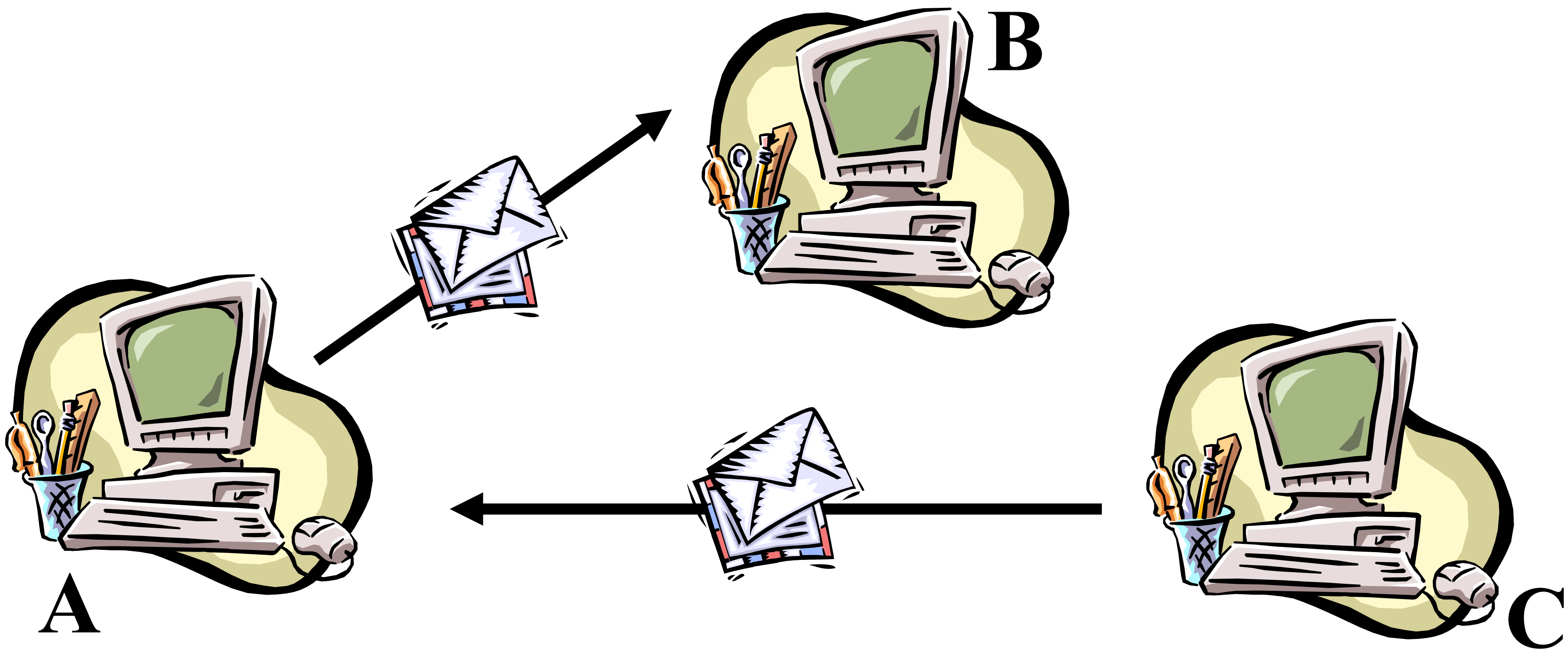
Results 4



Correctness

- Results 1: non-regular register (safe)
- Results 2; 3; 4: regular register

Message passing model



Implementing a register

- Implementing ***Read()*** and ***Write()*** operations at every process

- Before returning a ***Read()*** value, or returning the ok of a ***Write()***, the process must communicate with other processes

A fail-stop algorithm

- We assume a ***fail-stop*** model:
  - processes can fail by crashing (no recovery)
  - channels are reliable
  - failure detection is perfect (completeness and accurary)

A fail-stop algorithm

- We implement a ***regular*** register
  - every process pi has a local copy of the register value vi
  - every process reads ***locally***
  - the writer writes ***globally,*** i.e., at all (non-crashed) processes

A fail-stop algorithm

- Write(v) at pi
  - send [W,v] to all
  - for every pj, wait until either:
    - receive [ack] or
    - detect [pj]
  - Return ok

- At pi:

  when receive [W,v] from pj

  vi := v

  send [ack] to pj

- Read() at pi
  - Return vi

Correctness (liveness)

- ✓ A Read() is local and eventually returns
- ✓ A Write() eventually returns, by the
  - (a) the completeness property of the failure detector, and
  - (b) the reliability of the channels

Correctness (safety – 1)

- (a) In the absence of concurrent or failed operation, a Read() returns the last value written
  - Assume a Write(x) terminates and no other Write() is invoked. By the accuracy property of the failure detector, the value of the register at all processes that did not crash is x. Any subsequent Read() invocation by some process pj returns the value of pj, i.e., x, which is the last written value

Correctness (safety – 2)

- (b) A Read() returns the value concurrently written or the last value written
  - Let x be the value returned by a Read(): by the properties of the channels, x is the value of the register at some process. This value does necessarily come from the last or a concurrent Write().

But

- What if failure detection is not perfect
- Can we devise an algorithm that implements a regular register and tolerates an arbitrary number of crash failures?

Lower bound

- ***Proposition:*** any wait-free asynchronous implementation of a regular register requires a majority of correct processes

- Proof (sketch): assume a Write(v) is performed and n/2 processes crash, then a Read() is performed and the other n/2 processes are up: the Read() cannot see the value v

- The impossibility holds even with a 1-1 register (one writer and one reader)

The majority algorithm [ABD95]

- We assume that p1 is the writer and any process can be reader
- We assume that a majority of the processes is correct (the rest can fail by crashing – no recovery)
- We assume that channels are reliable
- Every process pi maintains a local copy of the register vi, as well as a sequence number sni and a read timestamp rsi
- Process p1 maintains in addition a timestamp ts1

Algorithm - Write()

- Write(v) at p1
  - ✓ ts1++
  - ✓ send [W,ts1,v] to all
  - ✓ when receive [W,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

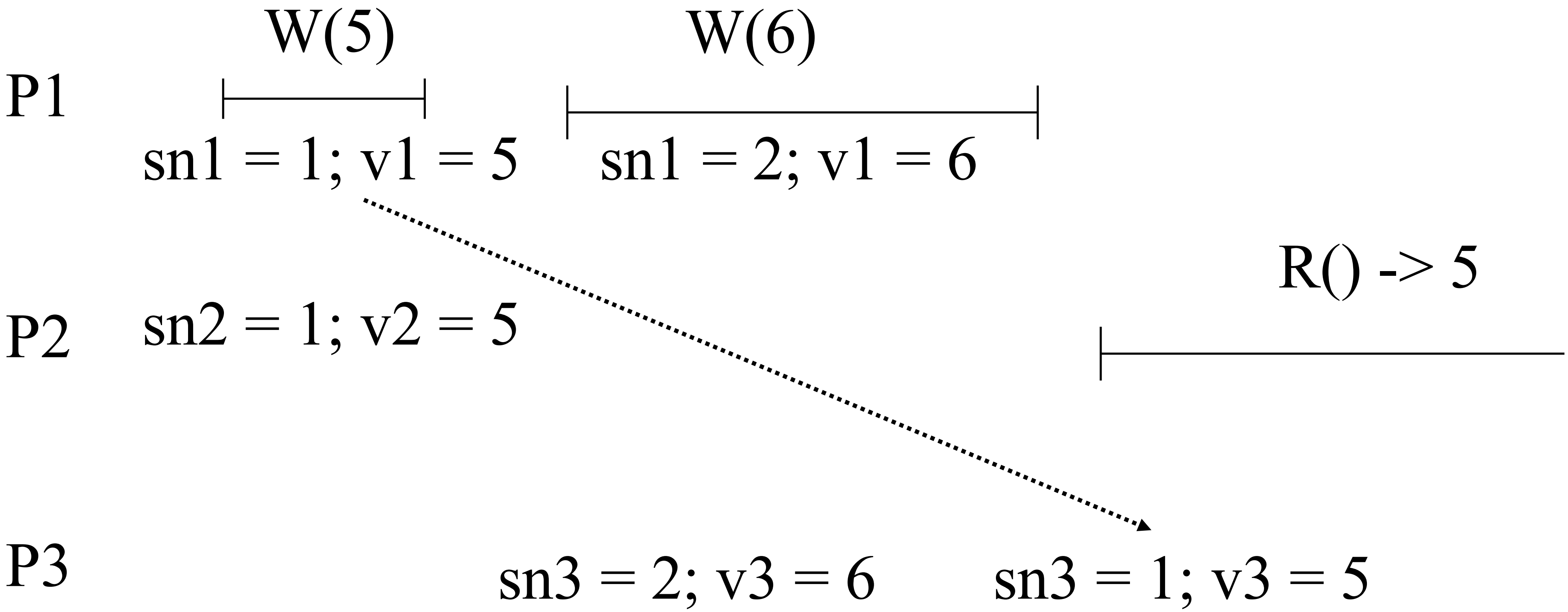
Algorithm - Read()

- Read() at pi
  - ✓ rsi++
  - ✓ 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

What if?

- Any process that receives a write message (with a timestamp and a value) updates its value and sequence number, i.e., without checking if it actually has an older sequence number

Old writes



Correctness 1

- ✓ Liveness: Any ***Read()*** or ***Write()*** eventually returns by the assumption of a majority of correct processes (if a process has a newer timestamp and does not send [W,ts1,ack], then the older Write() has already returned)

- ✓ Safety 2: By the properties of the channels, any value read is the last value written or the value concurrently written

Correctness 2 (safety – 1)

- (a) In the absence of concurrent or failed operation, a ***Read()*** returns the last value written
  - Assume a Write(x) terminates and no other Write() is invoked. A majority of the processes have x in their local value, and this is associated with the highest timestamp in the system. Any subsequent Read() invocation by some process pj returns x, which is the last written value

Atomicity

- ***An atomic register*** provides strong guarantees even when there is concurrency and failures: the execution is equivalent to a sequential and failure-free execution (***linearization***)

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

And

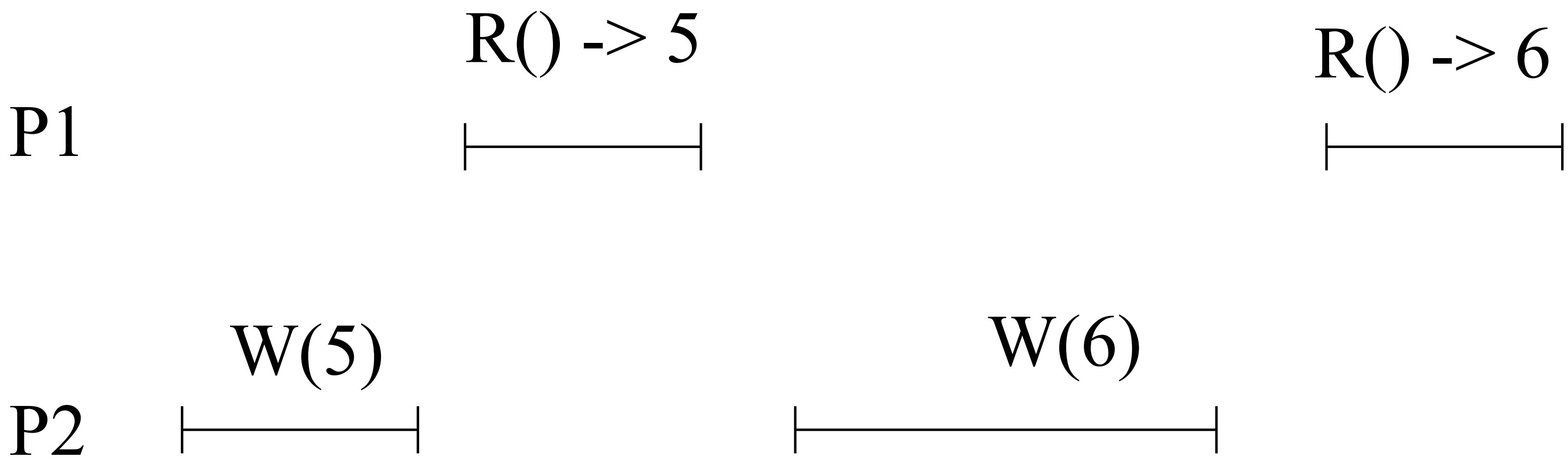
- Every complete operation appears to be executed at some instant between its invocation and reply time events

Regular vs Atomic

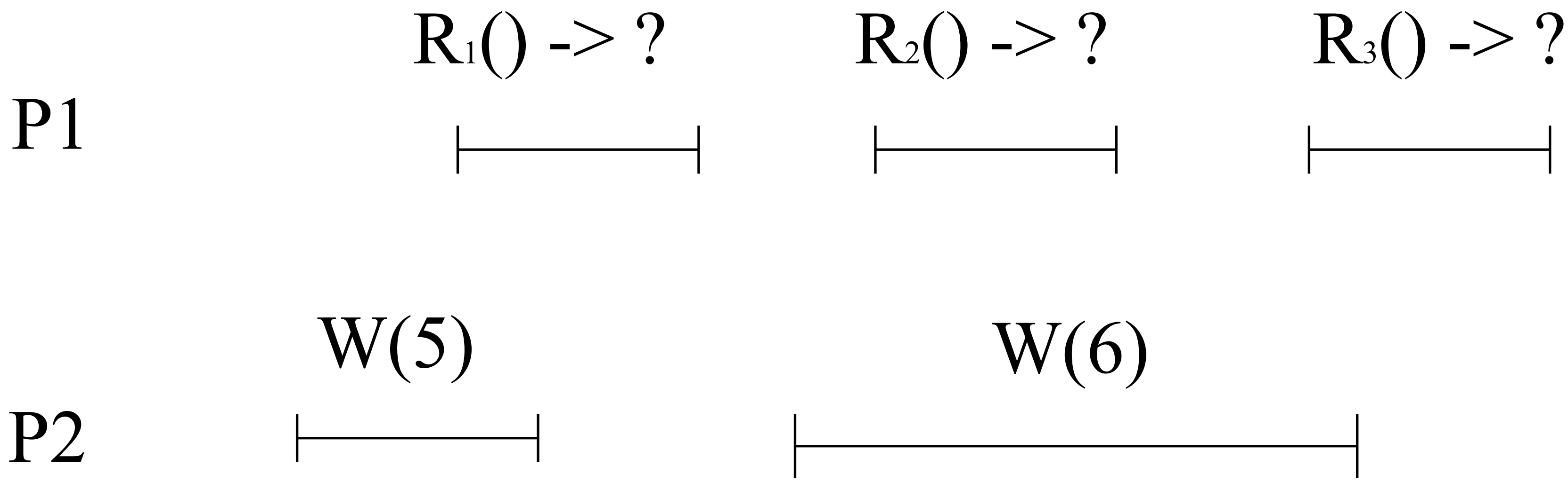
- For a regular register to be atomic, two successive ***Read()*** must not overlap a ***Write()***

- The regular register might in this case allow the first ***Read()*** to obtain the new value and the second ***Read()*** to obtain the old value

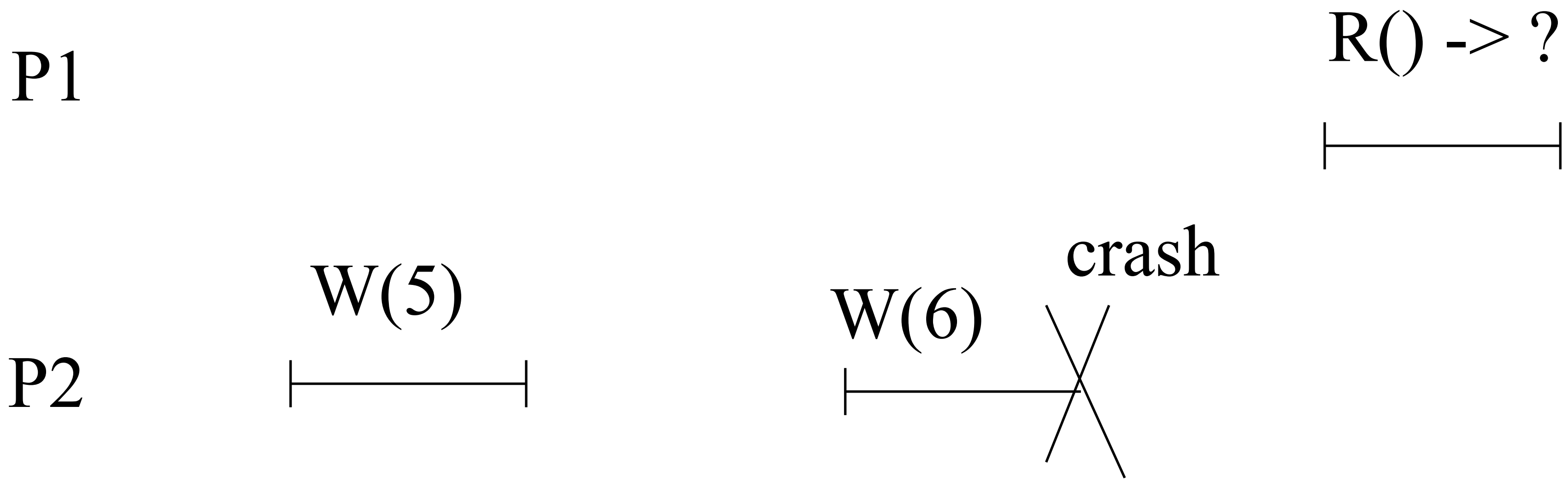
Sequential execution



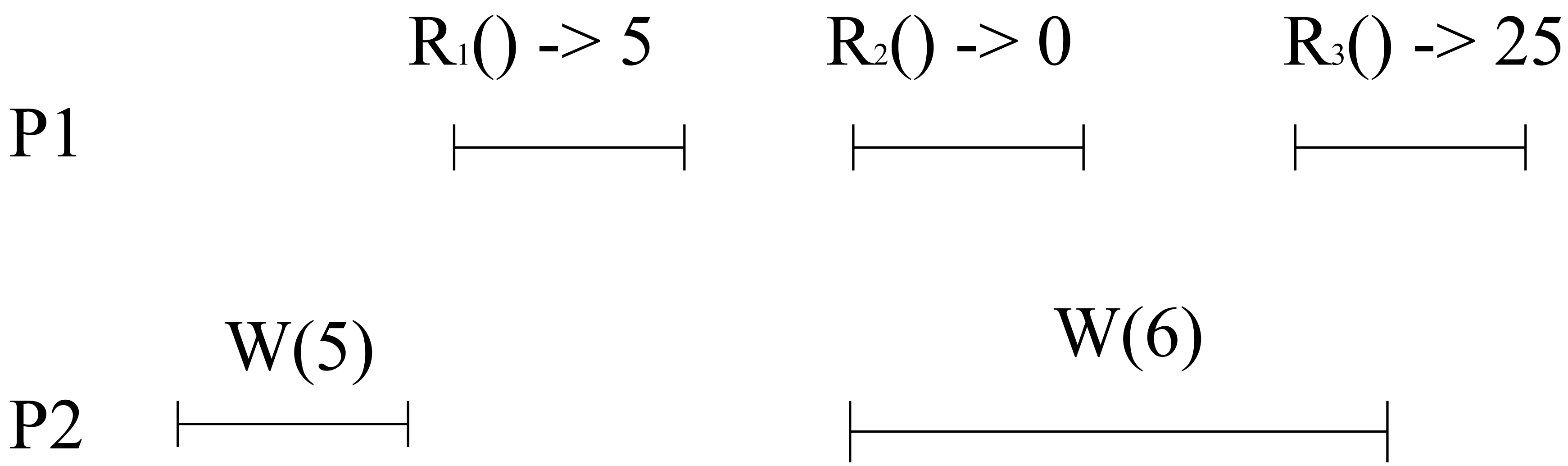
Concurrent execution



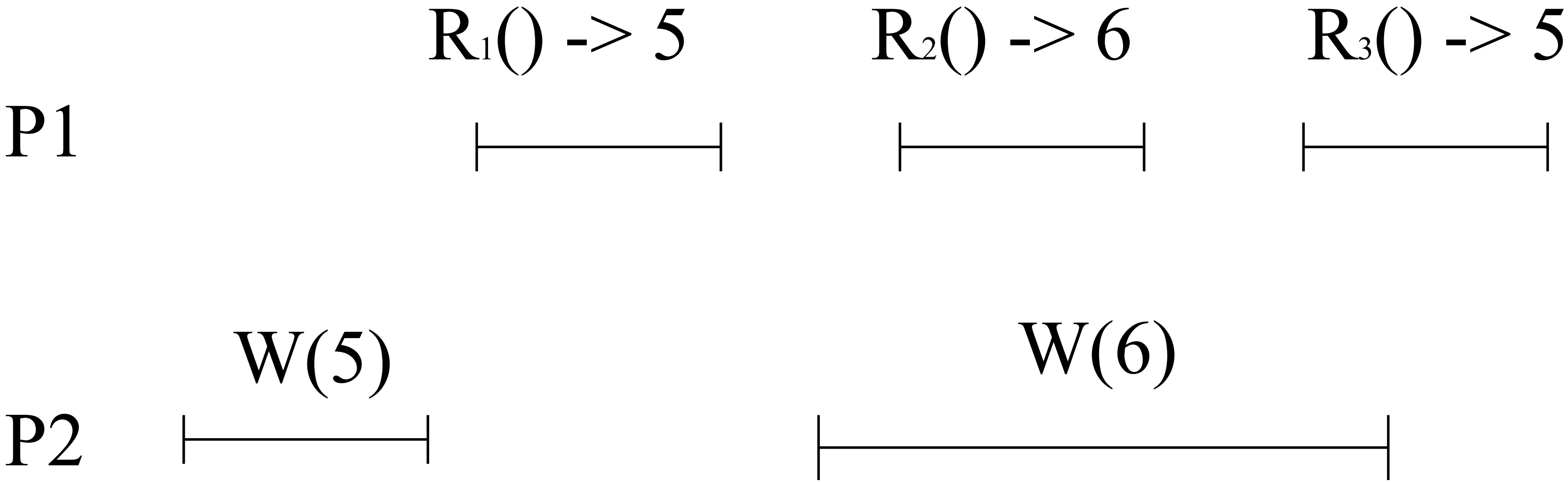
Execution with failures



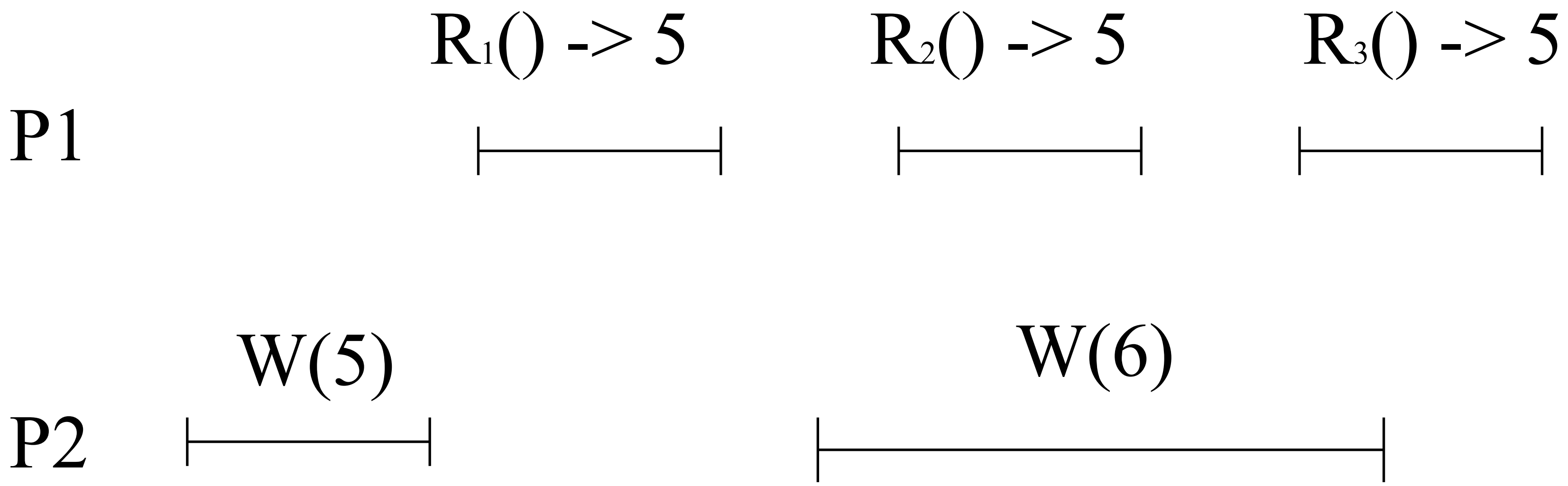
Execution 1



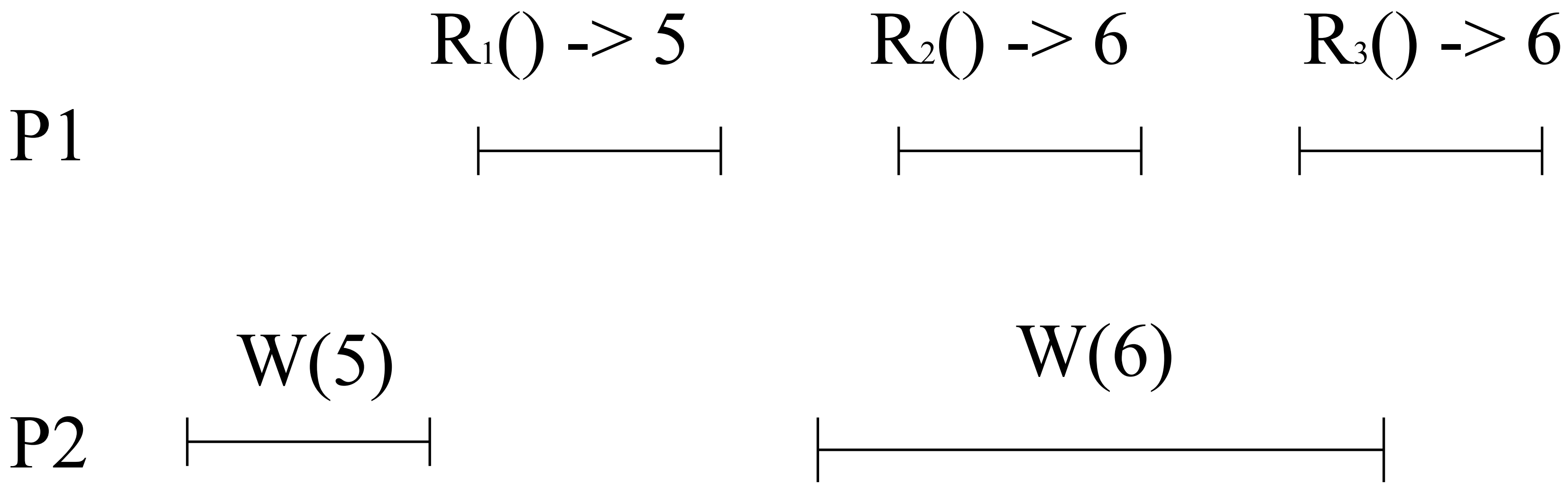
Execution 2



Execution 3

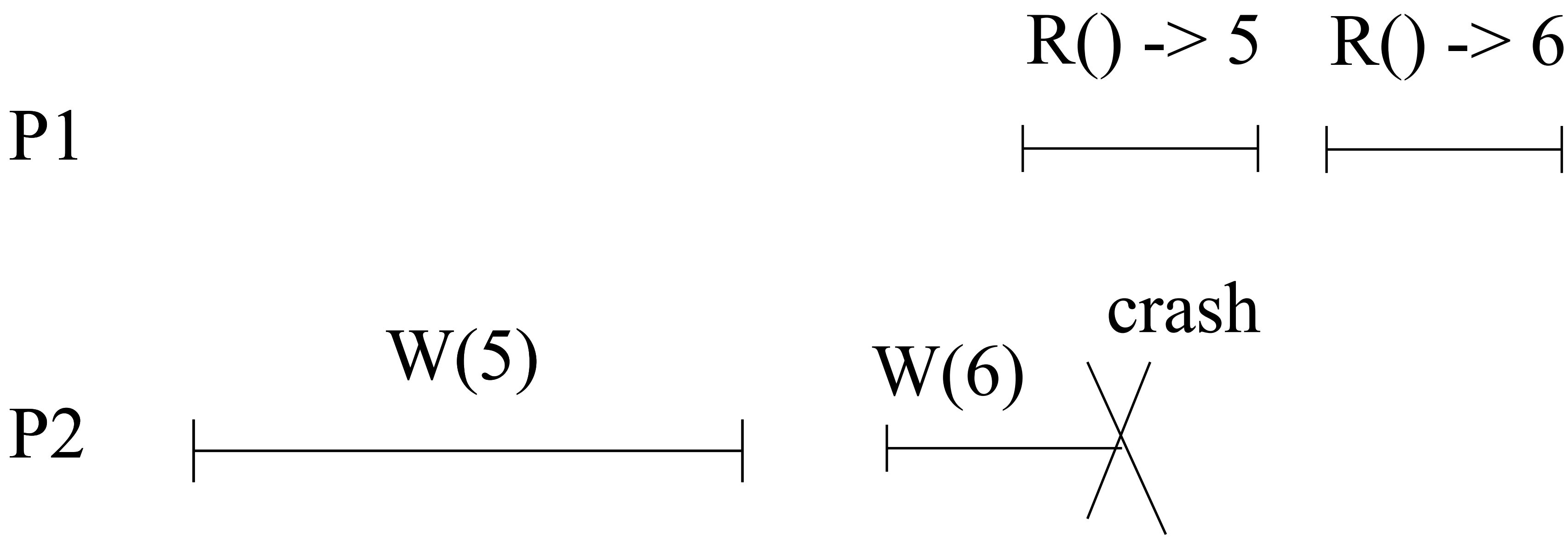


Execution 4

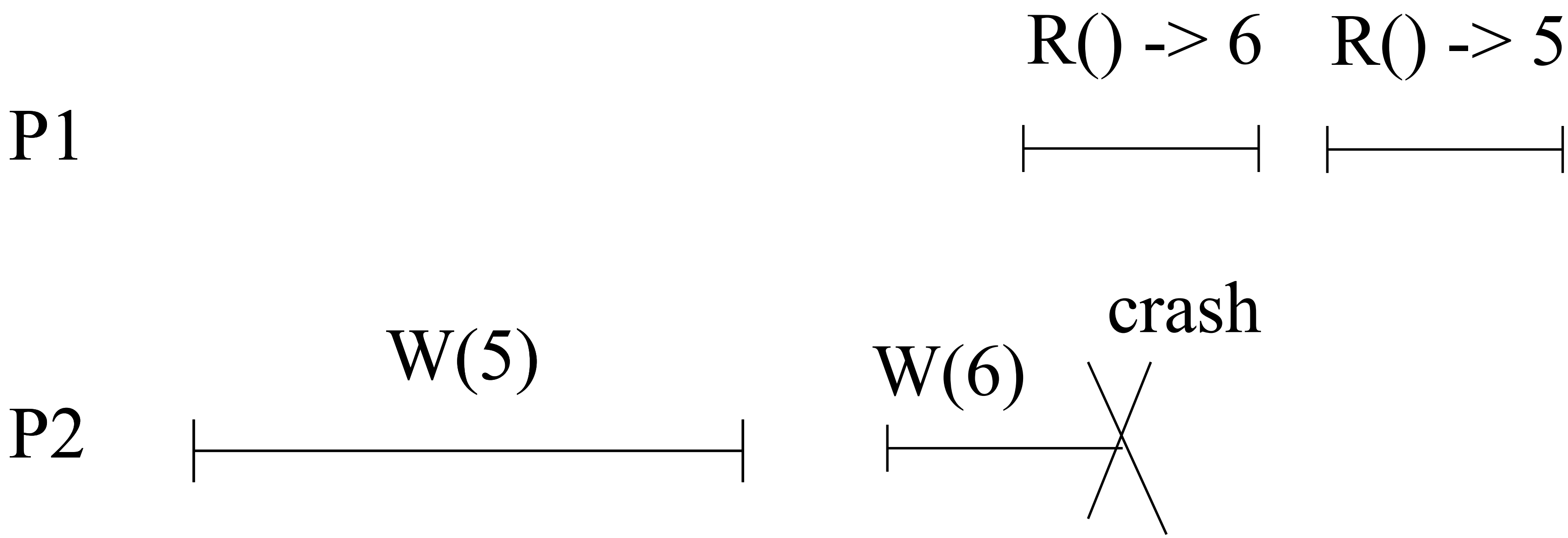


Execution 5

Execution 6



Execution 7



Fail-stop algorithms

- We first assume a fail-stop model; more precisely:
  - any number of processes can fail by crashing (no recovery)
  - channels are reliable
  - failure detection is perfect: accuracy and completeness

The regular algorithm

- Consider our fail-stop ***regular*** register algorithm
  - every process has a local copy of the register value
  - every process reads ***locally***
  - the writer writes ***globally,*** i.e., at all (non-crashed) processes

The regular algorithm

- Write(v) at pi
  - send [W,v] to all
  - for every pj, wait until either:
    - received [ack] or
    - detect [pj]
  - Return ok

- At pi:

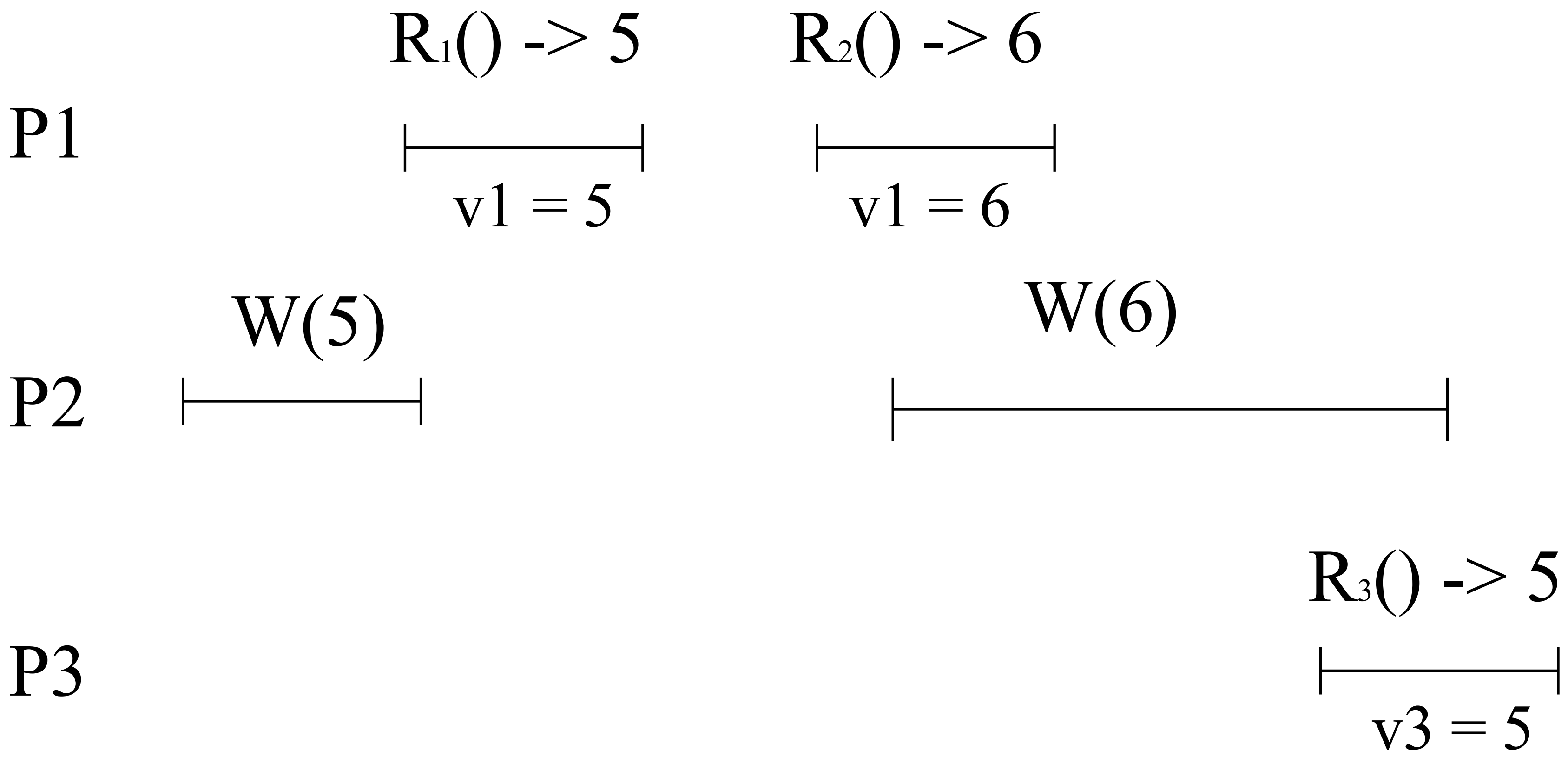
  when receive [W,v] from pj

  vi := v

  send [ack] to pj

- Read() at pi
  - Return vi

Atomicity?



Linearization?

$R_1()$ -> 5 | $R_2()$ -> 6

P1

W(5) | W(6)

P2

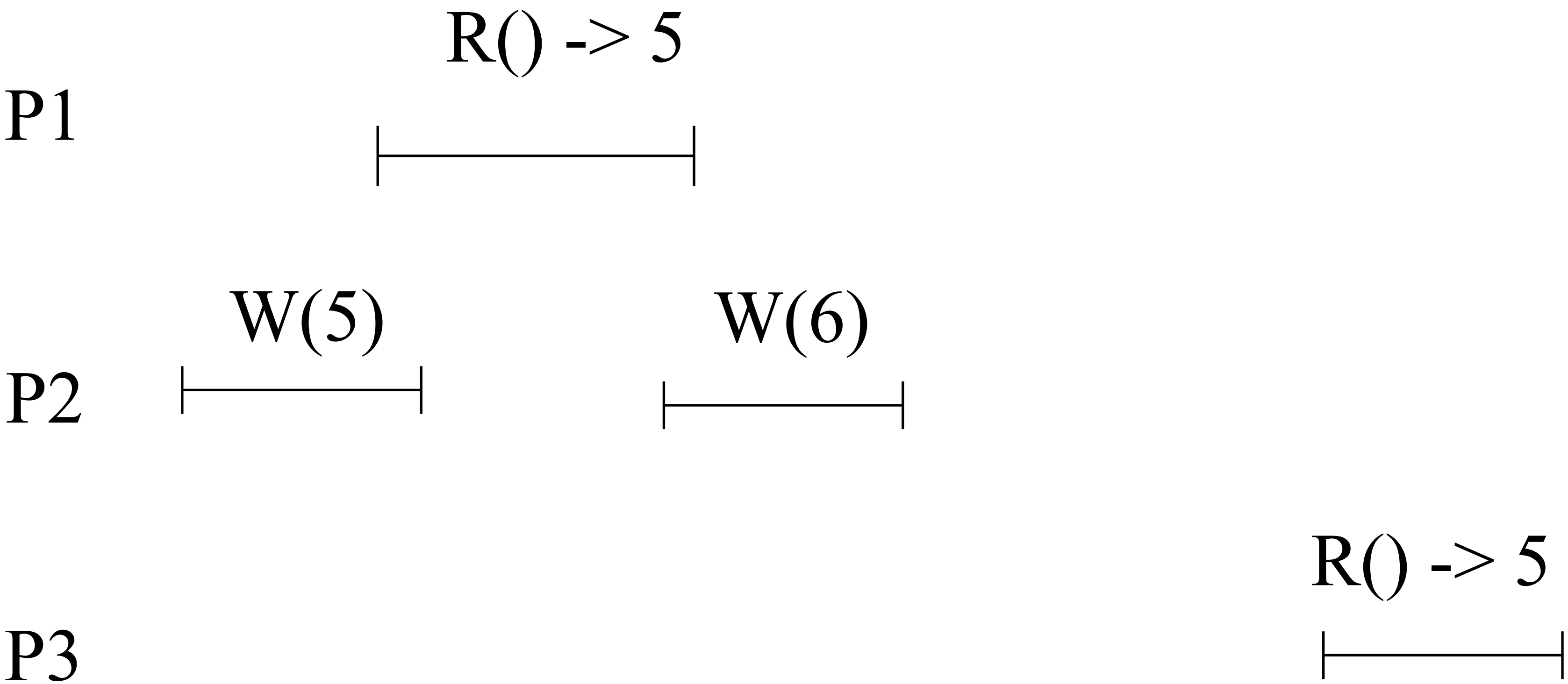
$R_3()$ -> 5 ??

P3

Fixing the pb: read-globally

- Read() at pi
  - send [W,vi] to all
  - for every pj, wait until either:
    - receive [ack] or
    - detect [pj]
  - Return vi

Still a problem



Linearization?

R_1() -> 5

P1

W(5) W(6)

P2

R_3() -> 5 ??

P3

A fail-stop 1-1 atomic algorithm

- Write(v) at p1
  - send [W,v] to p2
  - Wait until either:
    - receive [ack] from p2 or
    - detect [p2]
  - Return ok

- At p2:

  when receive [W,v] from p1

  v2 := v

  send [ack] to p2

- Read() at p2
  - Return v2

A fail-stop 1-N algorithm

- every process maintains a local value of the register as well as a sequence number
- the writer, p1, maintains, in addition a timestamp ts1
- any process can read in the register

A fail-stop 1-N algorithm

- Write(v) at p1
  - ts1++
  - send [W,ts1,v] to all
  - for every pi, wait until either:
    - receive [ack] or
    - detect [pi]
  - Return ok

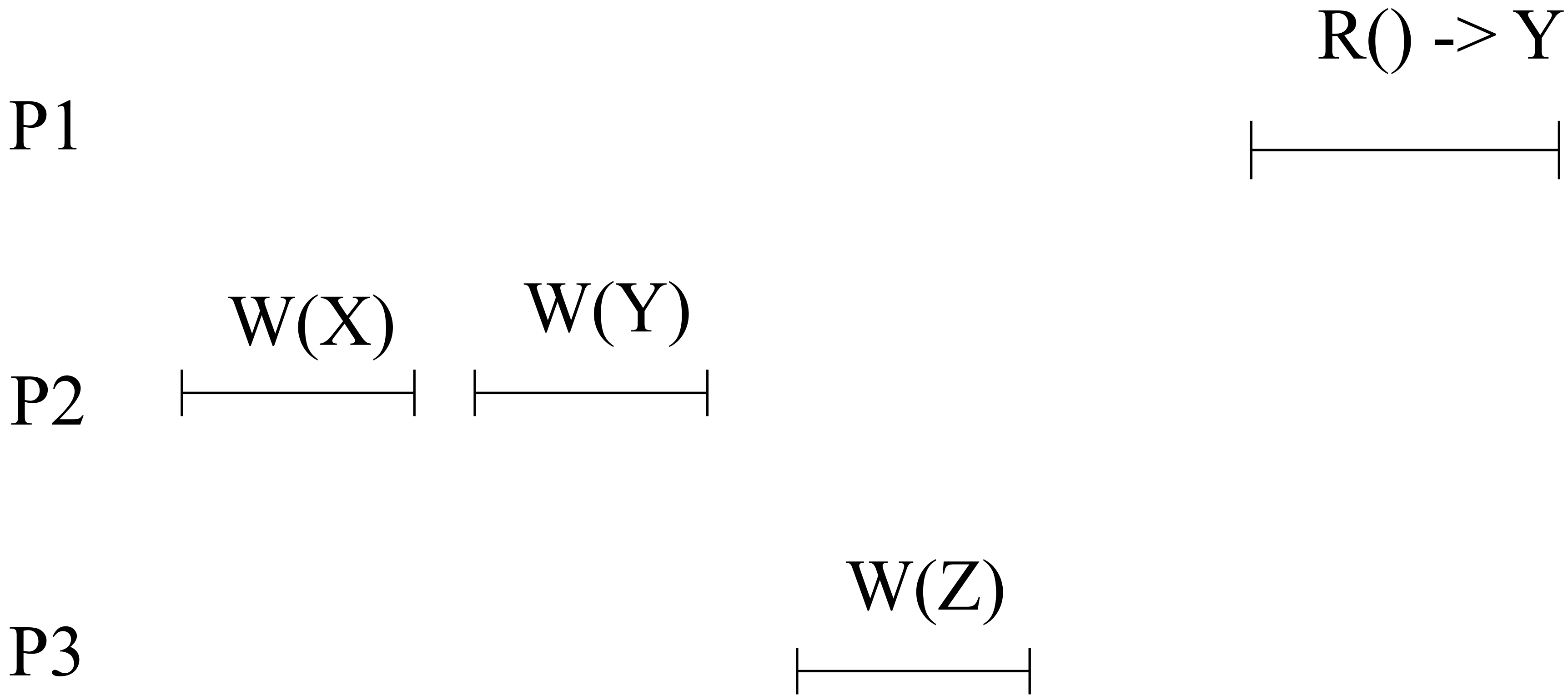
- Read() at pi
  - send [W,sni,vi] to all
  - for every pj, wait until either:
    - receive [ack] or
    - suspect [pj]
  - Return vi

A 1-N algorithm (cont'd)

- At pi
  - When pi receive [W,ts,v] from pj

```
if ts > sni then
    vi := v
    sni := ts
send [ack] to pj
```

Why not N-N?



The Write() algorithm

- Write(v) at pi
  - ✓ send [W] to all
  - ✓ for every pj wait until
    - **receive [W,snj] or**
    - **suspect pj**
  - ✓ (sn,id) := (highest snj + 1,i)
  - ✓ send [W,(sn,id),v] to all
  - ✓ for every pj wait until
    - **receive [W,(sn,id),ack] or**
    - **detect [pj]**
  - ✓ Return ok

- At pi

T1:

- ✓ when receive [W] from pj
  - send [W,sn] to pj

T2:

- ✓ when receive [W,(snj,idj),v] from pj
- ✓ If (snj,idj) > (sn,id) then
  - vi := v
  - (sn,id) := (snj,idj)
- ✓ send [W,(snj,idj),ack] to pj

The Read() algorithm

- Read() at pi
  - ✓ send [R] to all
  - ✓ for every pj wait until
    - **receive [R,(snj,idj),vj] or**
    - **suspect pj**
  - ✓ v = vj with the highest (snj,idj)
  - ✓ (sn,id) = highest (snj,idj)
  - ✓ send [W,(sn,id),v] to all
  - ✓ for every pj wait until
    - **receive [W,(sn,id),ack] or**
    - **detect [pj]**
  - ✓ Return v

- At pi

  T1:
  - ✓ when receive [R] from pj
    - send [R,(sn,id),vi] to pj

  T2:
  - ✓ when receive [W,(snj,idj),v] from pj
  - ✓ If (snj,idj) > (sn,id) then
    - vi := v
    - (sn,id) := (snj,idj)
  - ✓ send [W,(snj,idj),ack] to pj

From fail-stop to fail-silent

- We assume a majority of correct processes
- In the 1-N algorithm, the writer writes in a majority using a timestamp determined locally and the reader selects a value from a majority and then imposes this value on a majority
- In the N-N algorithm, the writers determines first the timestamp using a majority