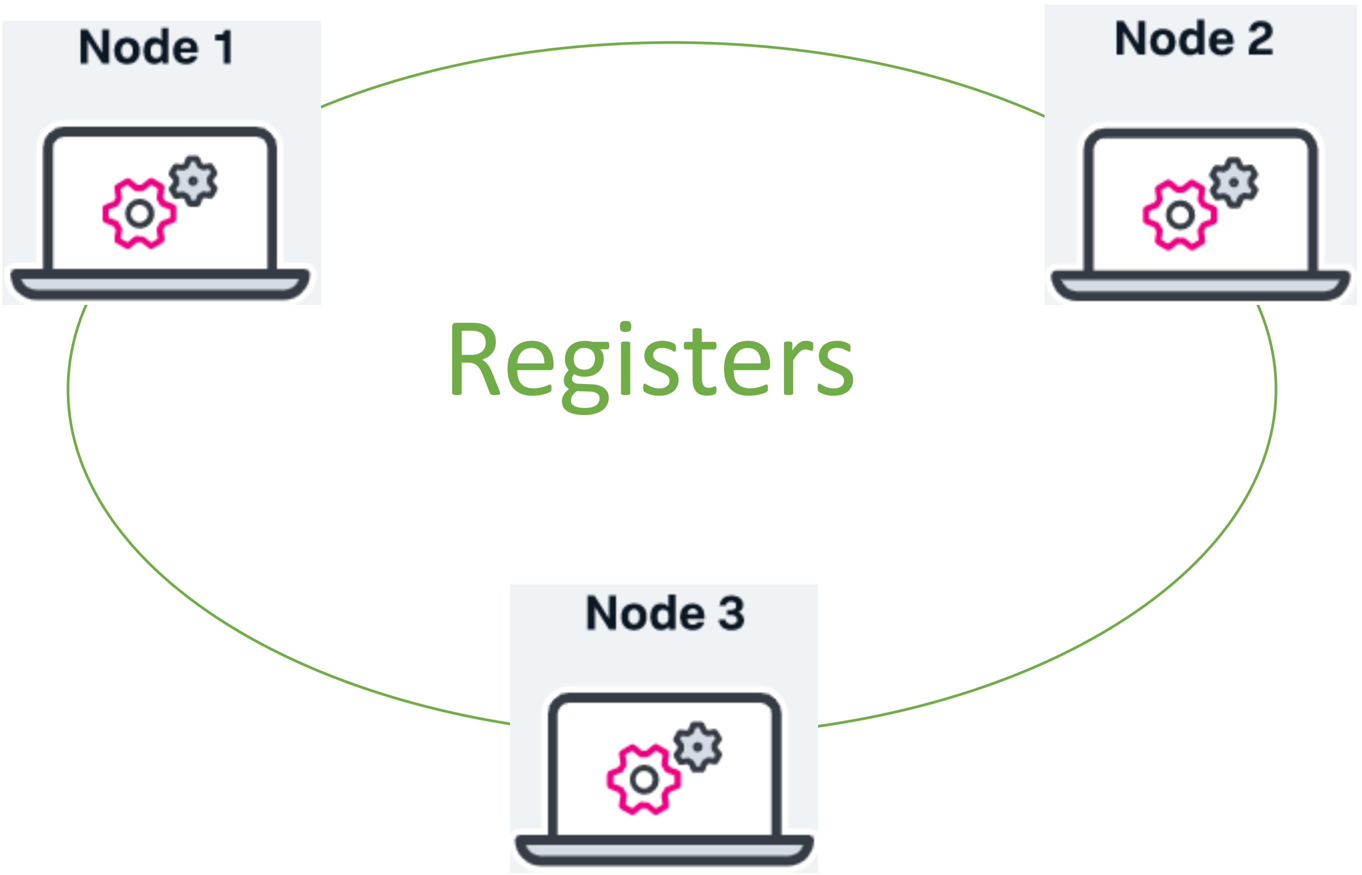
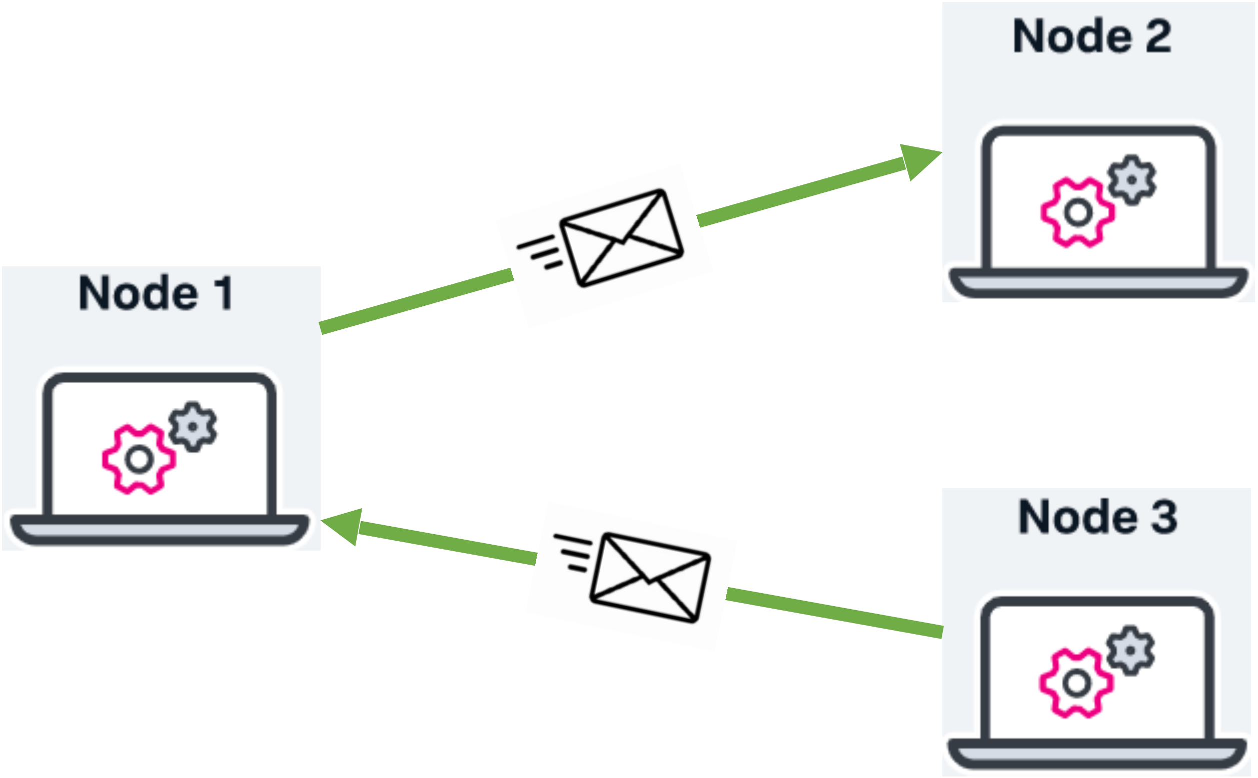
Simulating Shared Memory

Mohsen Lesani

The application model



Message passing



Register (assumptions)

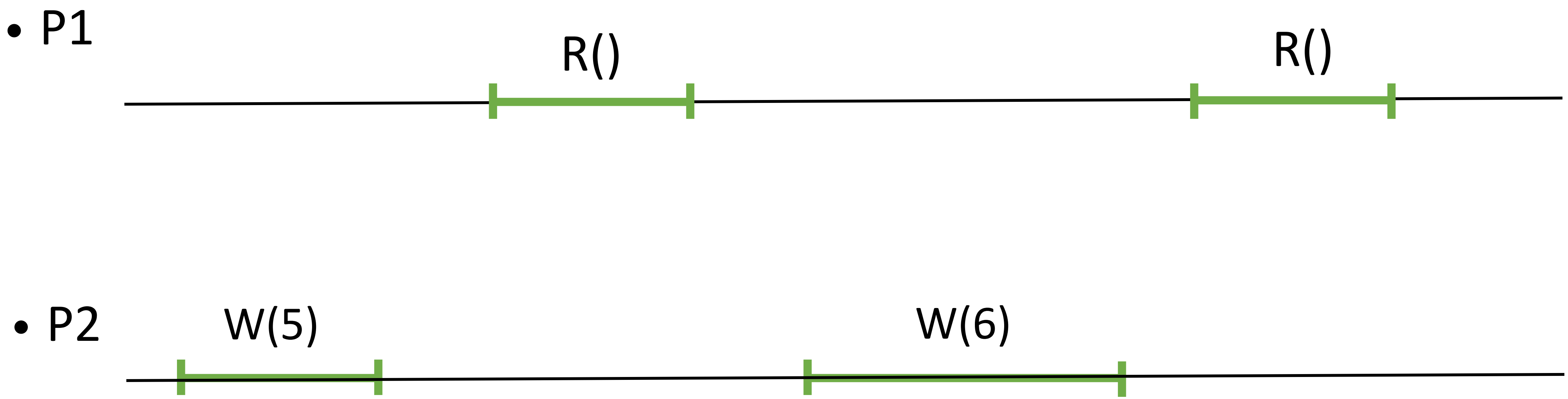
- For presentation simplicity, we assume registers of **integers**.
- We also assume that the initial value of a register is 0 and this value is initialized (write()en) by some process before the register is used
- We assume that every value written is **uniquely** identified (this can be ensured by associating a process id and a timestamp with the value)

Register Specification

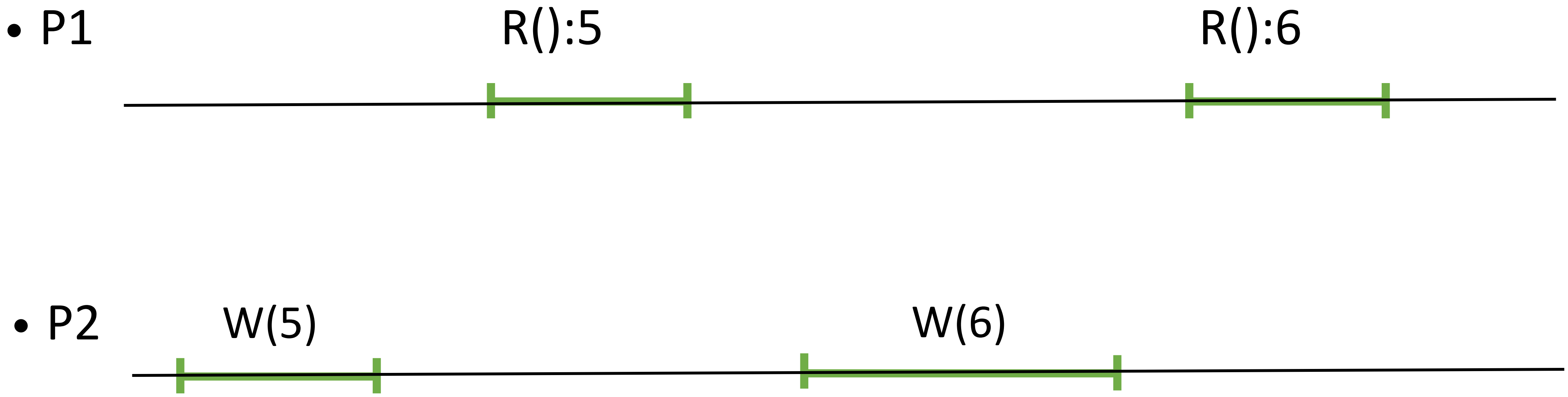
- Assume a register that is local to a process, i.e., accessed only by one process:

- In this case, the value returned by a **Read()** is the last value **Write()**en.

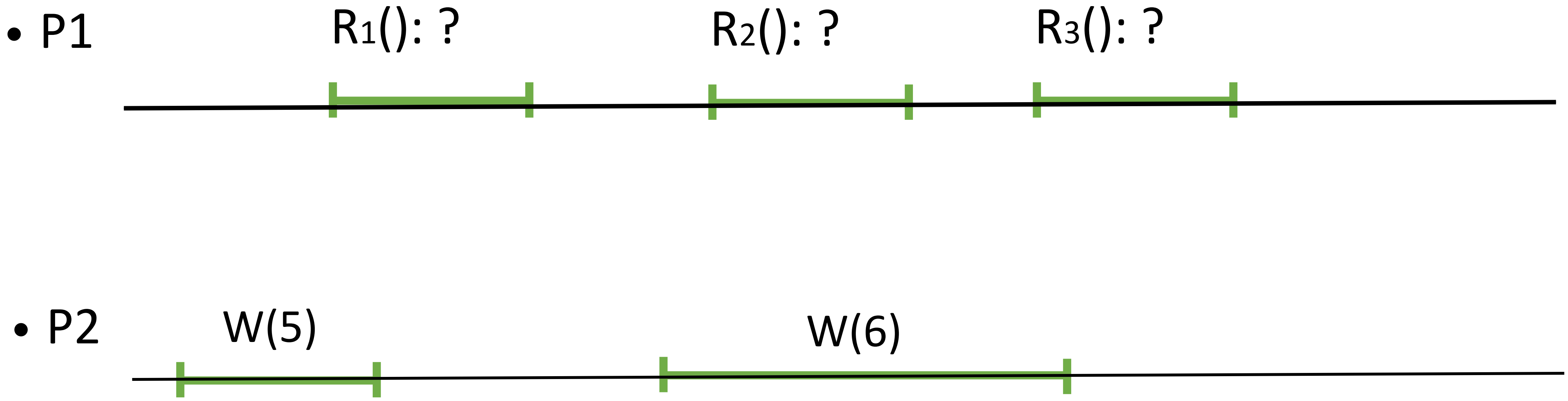
Sequential execution



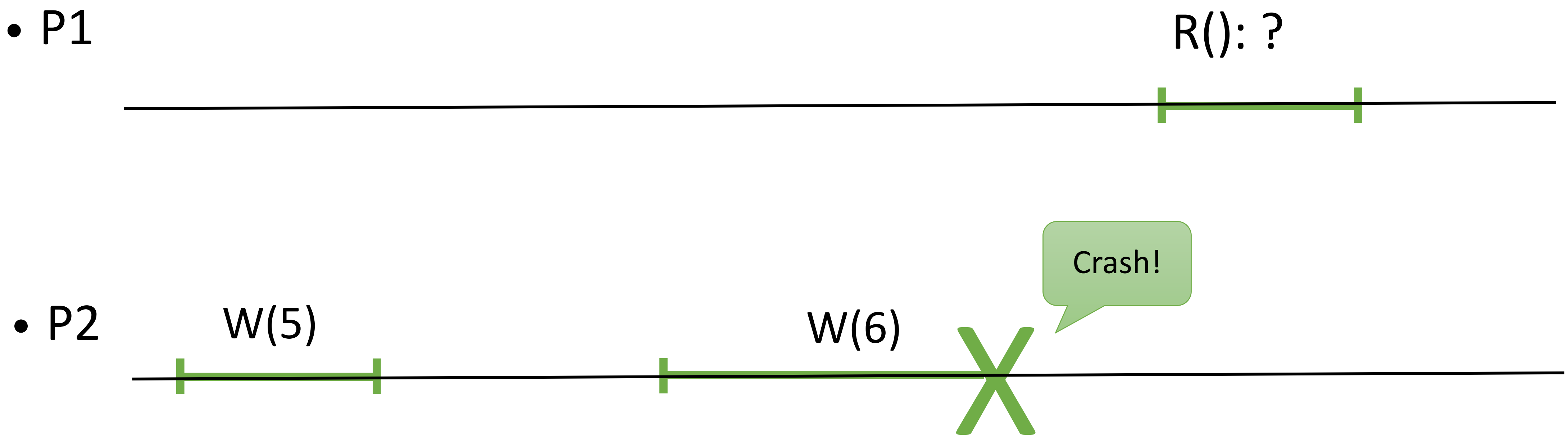
Sequential execution



Concurrent execution



Concurrent execution



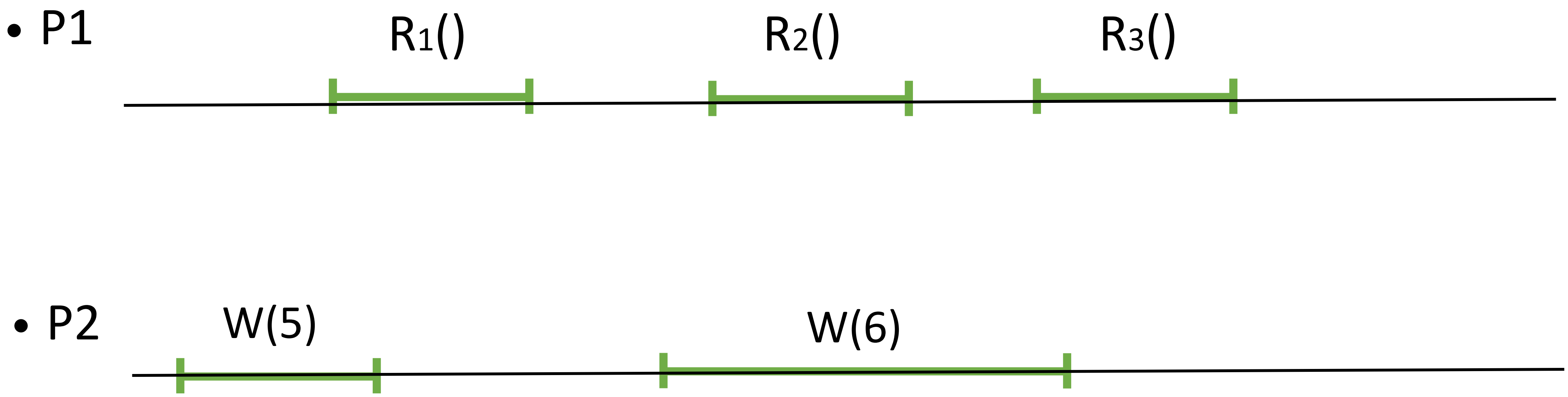
Regular register

- It assumes only **one** writer; multiple processes might however read from the register.
- It provides **strong** guarantees when there is no concurrent or failed operations (invoked by processes that fail in the middle)
- When some operations are concurrent, or some operation fails, the register provides **minimal** guarantees

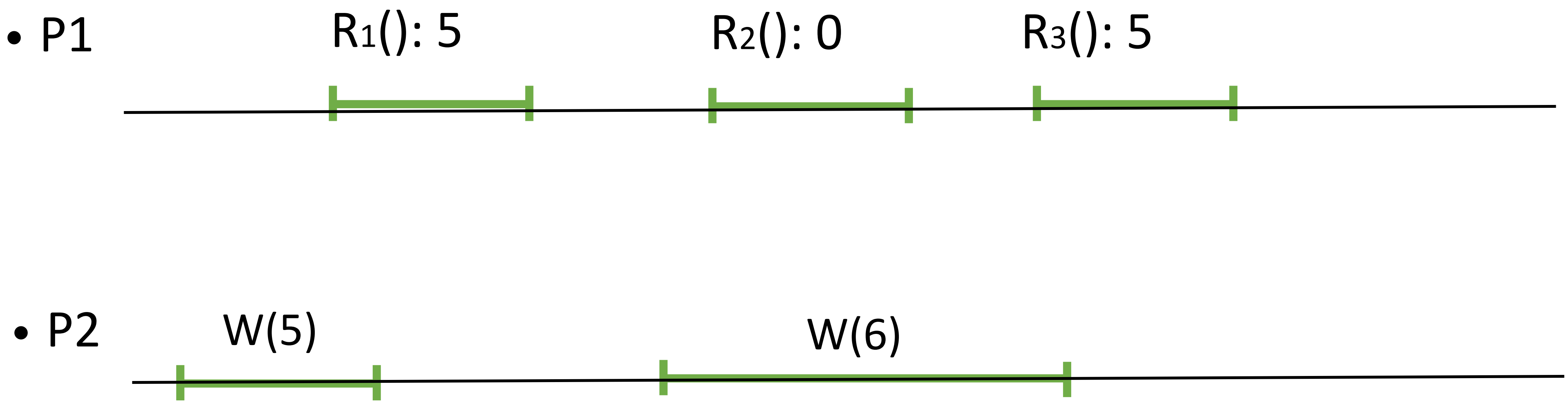
Regular register

- **Read()** returns:
  - The last value written if there is no concurrent or failed operations.
  - Otherwise the last value **Write()**en or **any** value concurrently **Write()**en.

Execution

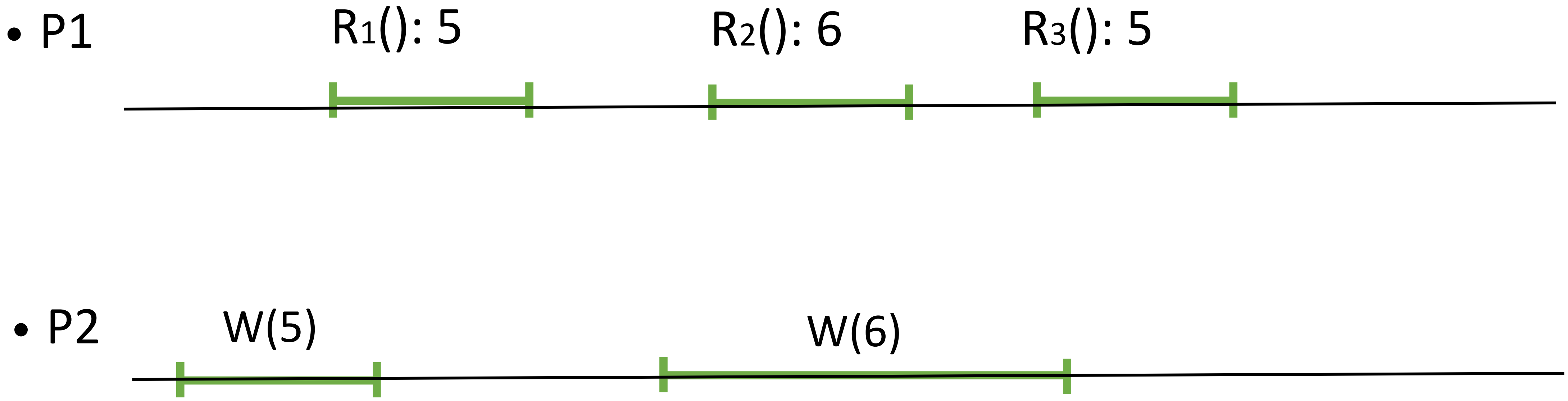


Results 1



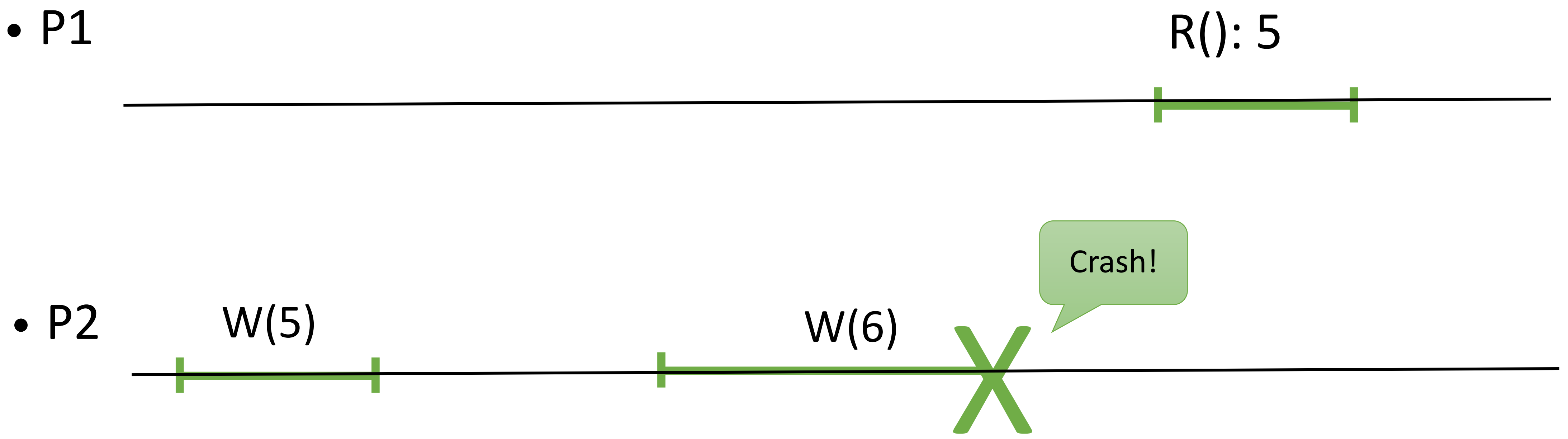
Not regular. The return values of R_2 is incorrect. (This is the so-called safe execution that is ironically not so safe.)

Results 2



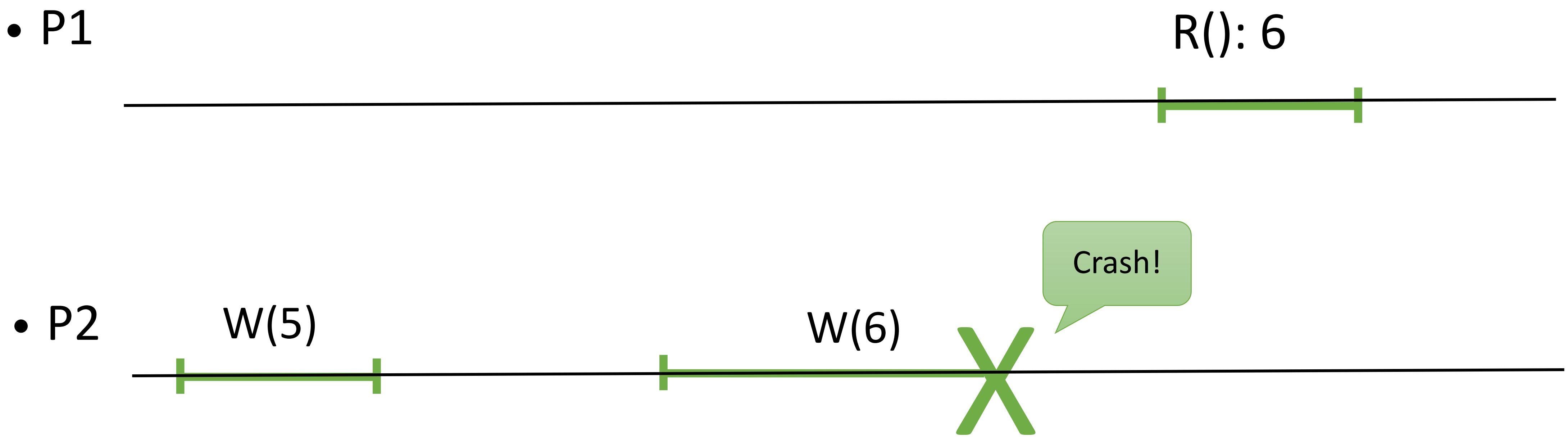
This is regular. R_2 returns the concurrently written value and R_3 returns the last written value.

Results 3



Regular. R written the last written value.
W(6) is like a concurrent write that never finishes.

Results 4



Regular. R returns the value written by the crashed write.

Regular Register Algorithms

Overview of this lecture

1. **Overview of a register algorithm**
2. A bogus algorithm
3. A simplistic algorithm
4. A simple fail-stop algorithm
5. A tight asynchronous lower bound
6. A fail-silent algorithm

Implementing a register

Implementing the register comes down to implementing **Read**() and **Write**() operations at every process

Implementing a register

- Before returning a **Read**() value, the process must communicate with other processes.

- Before finishing a **Write**(), i.e., returning the corresponding ok, the process must communicate with other processes.

Overview of this lecture

1. Overview of a register algorithm
2. **A bogus algorithm**
3. A simplistic algorithm
4. A simple fail-stop algorithm
5. A tight asynchronous lower bound
6. A fail-silent algorithm

A Bogus Algorithm

- We assume that channels are reliable (perfect point-to-point links)
- Every process p_i holds a copy of the register value v_i

A Bogus Algorithm

upon Read() at p_i

trigger Ret(v_i)

upon Write(v) at p_i

v_i := v

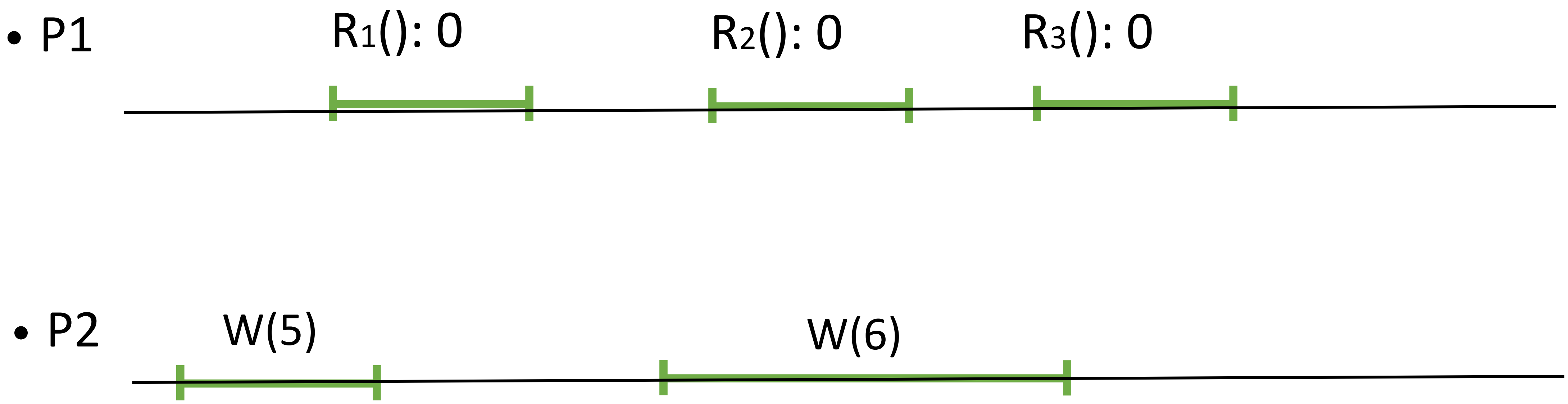
trigger ok

The resulting register is live but not safe:

Even in a sequential and failure-free execution, a **Read**() by p_j might not return the last written value, say by p_i

A Bogus Algorithm

No Safety



Overview of this lecture

1. Overview of a register algorithm
2. A bogus algorithm
3. **A simplistic algorithm**
4. A simple fail-stop algorithm
5. A tight asynchronous lower bound
6. A fail-silent algorithm

A Simplistic Algorithm

- We still assume that channels are reliable but now we also assume that no process fails
- Basic idea: one process, say p_1, holds the value of the register

A Simplistic Algorithm

upon Read() at p_i

trigger send [R] to p_1

wait to receive [v]

trigger Ret(v)

upon Write(v) at p_i

trigger send [W,v] to p_1

wait to receive [ok]

trigger ok

At p_1:

upon deliver [R] from p_i

trigger send [v_1] to p_i

upon deliver [W,v] from p_i

v_1 := v

trigger send [ok] to p_i

Correctness (liveness)

- Wait-free: every request is eventually followed by a response.
- By the assumption that
  - no process fails
  - channels are reliable
- No wait statement blocks forever, and hence every invocation eventually terminates

Correctness (safety)

- If there is no concurrent or failed operation, a **Read**() returns the last value written.
  - Assume a Write(x) terminates and no other Write() is invoked. The value of the register is hence x at p_1. Any subsequent Read() invocation by some process p_j returns the value of p_1, *i.e.*, x, which is the last written value.

- A **Read**() returns the previous value written or the value concurrently written.
  - Let x be the value returned by a Read(). By the properties of the channels, x is the value of the register at p_1. This value has been obviously written by only the last or a concurrent Write().

What if?

- Processes might crash?
- If p_1 is always up, then the register is regular and wait-free.
- If p_1 crashes, then the register is not wait-free.
- The value cannot be hosted by only one process.

Overview of this lecture

1. Overview of a register algorithm
2. A bogus algorithm
3. A simplistic algorithm
4. **A simple fail-stop algorithm**
5. A tight asynchronous lower bound
6. A fail-silent algorithm

The fail-stop model

- We assume a fail-stop model: more precisely,
  - any number of processes can fail by crashing (no recovery)
  - failure detection is perfect (we have a perfect failure detector)
  - channels are reliable

Fail-stop N-N algorithm

- We implement a **regular** register
  - Every process can be reader and writer.
  - Every process p_i has a local copy of the register value v_i.
  - Every process reads **locally**.
  - The writer writes **globally**, i.e., at all (non-crashed) processes.

Fail-stop N-N algorithm

upon Write(v) at p_i

trigger send [W,v] to all

foreach p_j, wait until either:

deliver [ack] or

suspect [p_j]

trigger ok

At p_i :

upon deliver [W,v] from p_j

v_i := v

trigger send [ack] to p_j

upon Read() at p_i

trigger Ret(v_i)

Correctness (liveness)

- A Read() is local and eventually returns.
- A Write() eventually returns, by
  - The strong completeness property of the failure detector

    The protocol eventually does not wait for incorrect processes.
  - The reliability of the channels.

    Acknowledgments are received from correct processes.

Correctness (safety)

- 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 p_j returns the value of p_j, i.e., x, which is the last written value.

- A **Read**() returns the value concurrently written or the last value written.
  - Let x be the value returned by a Read() at process p_i. The value x is the stored value v_i of p_i. The stored value of a process has been written only by the last or a concurrent Write().

What if?

Failure detection is not perfect.

Overview of this lecture

1. Overview of a register algorithm
2. A bogus algorithm
3. A simplistic algorithm
4. A simple fail-stop algorithm
5. **A tight asynchronous lower bound**
6. A fail-silent algorithm

The fail-silent model

- We assume a **fail-silent model:**
  - Any number of processes can fail by crashing (no recovery)
  - There is no accurate failure detector.
  - Channels are reliable

Lower bound

- **Proposition**: Any wait-free asynchronous implementation of a regular register requires a majority (quorum) of processes to be correct.
- Proof (sketch):
  - Assume that this is possible with less than a correct majority. Assume a Write(v) is performed. In the absence of failure detectors, to guarantee liveness, this operation can write into and wait for at most $\lfloor n/2 \rfloor$ processes. (If failure detector was available, the process could wait to write to all non-crashed processes.) Since at most $\lfloor n/2 \rfloor$ of processes need to be correct, let the written processes crash and let others be correct. Then, a Read() is performed. The Read() cannot see the value v.
- The impossibility holds even with a 1-1 register (one writer and one reader)

Overview of this lecture

1. Overview of a register algorithm
2. A bogus algorithm
3. A simplistic algorithm
4. A simple fail-stop algorithm
5. A tight asynchronous lower bound
6. **A fail-silent algorithm**

The majority algorithm (Fail-silent 1-N)

Idea:

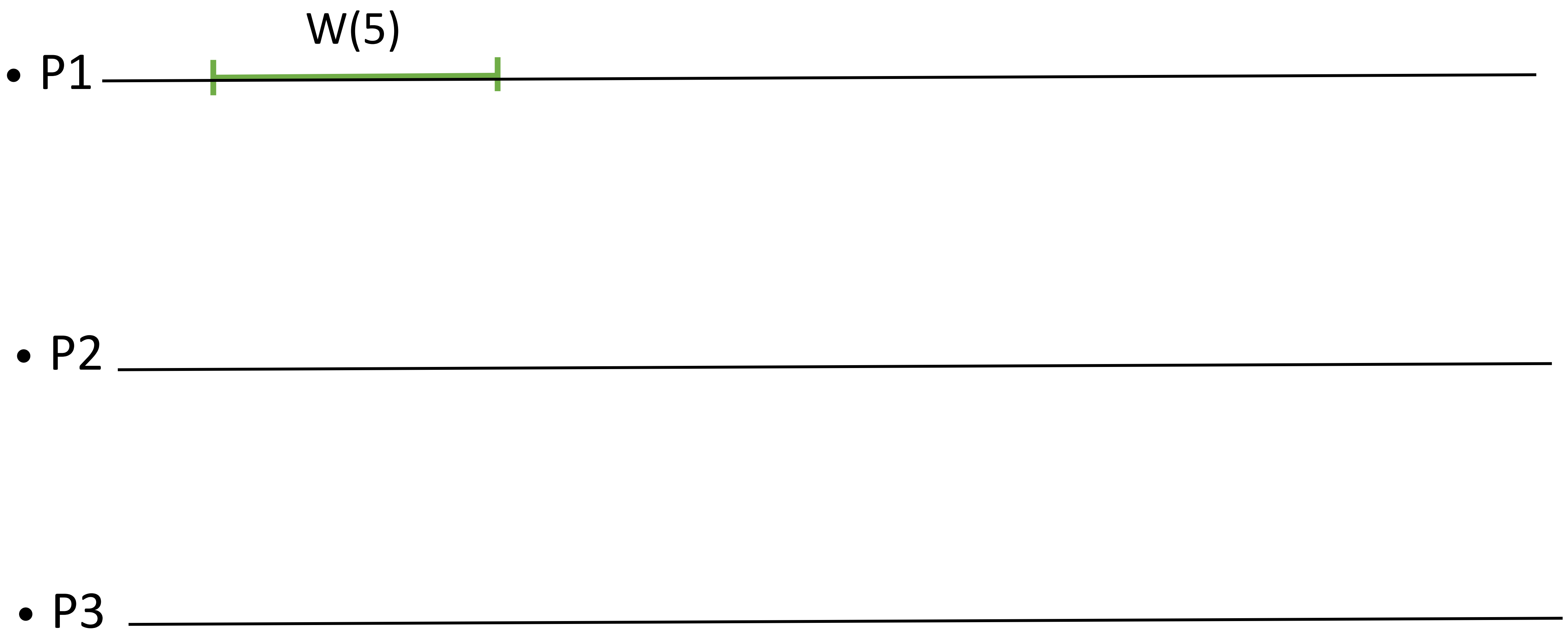
- On write, send the value and receive ack from a quorum (majority).
- On read, get the value from a quorum (majority) and return the newest value.

- To recognize the newest value, the writer maintains and propagates a timestamp.
- Each reader maintains a local timestamp, and sends and receives it to distinguish between responses to its different reads.

The majority algorithm (Fail-silent 1-N)

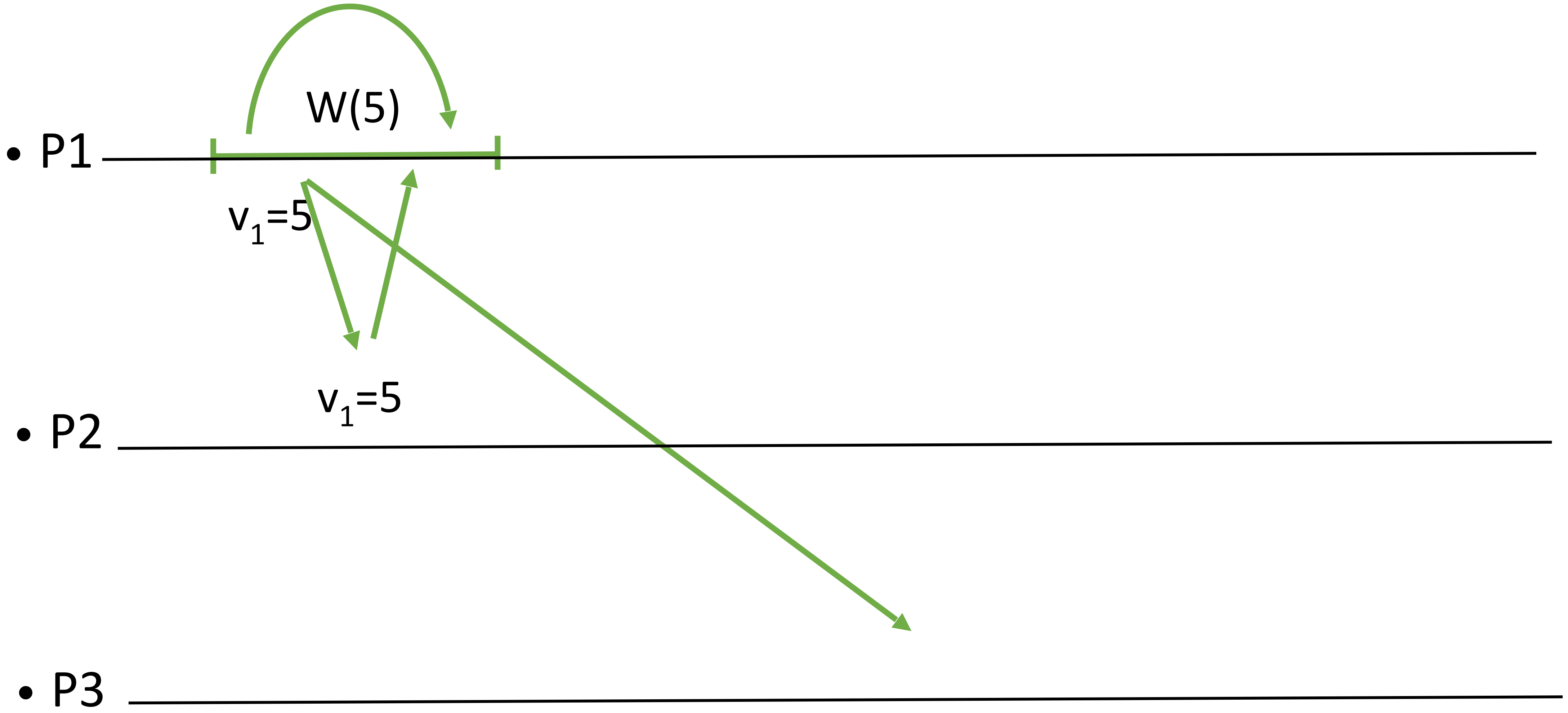
- We assume that p_1 is the writer and any process can be a reader.
- Every process p_i stores a local copy of the register v_i.
- The writer process p_1 maintains a timestamp ts_1 that is incremented on each write.
- Each process p_i stores the sequence number sn_i that is the timestamp of its stored value v_i.
- Each process p_i stores the read timestamp rs_i that is a local timestamp in p_i to distinguish its Read() operations.

Old Writes



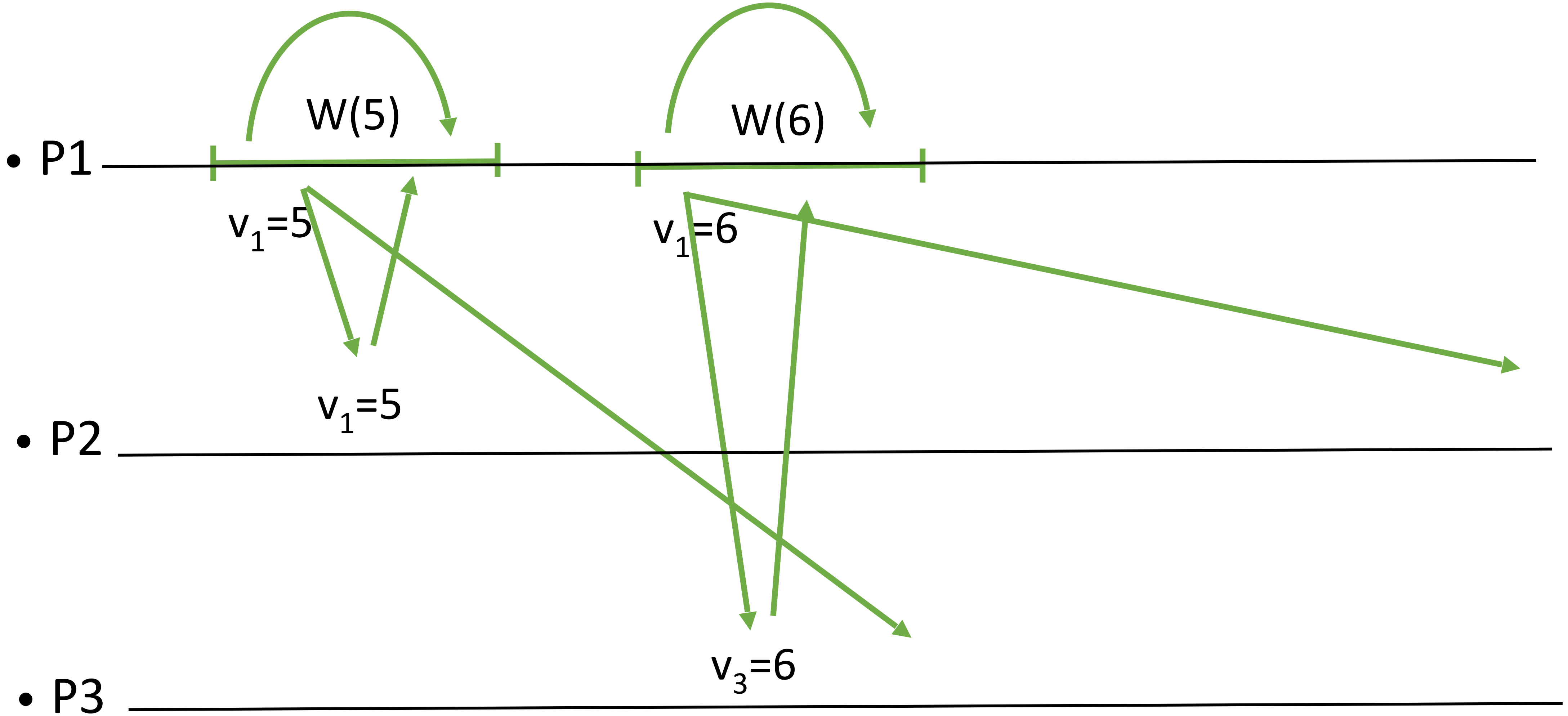
Incorrect execution. In p_3, the write message with $sn_1 = 1$ should be ignored.

Old Writes



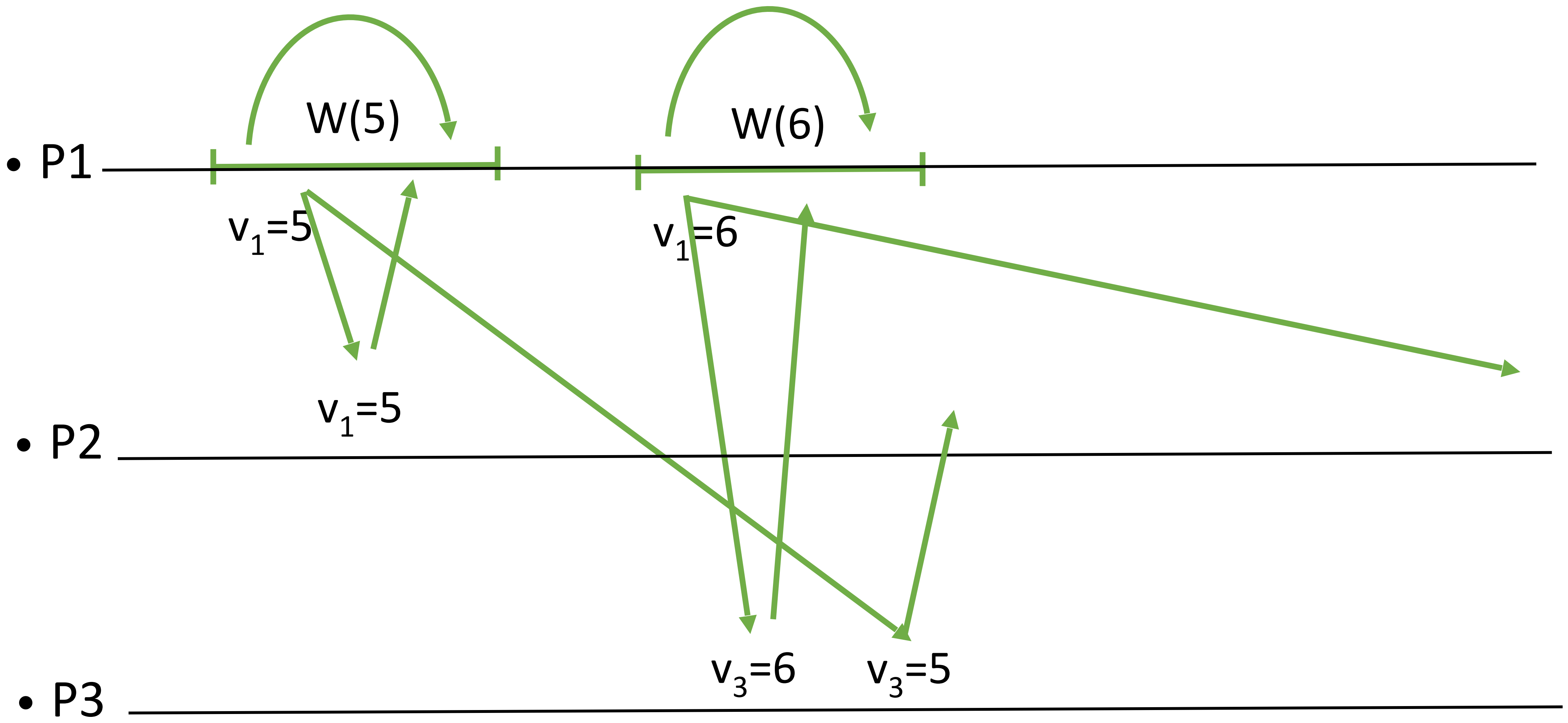
Incorrect execution. In p_3, the write message with sn_1 = 1 should be ignored.

Old Writes



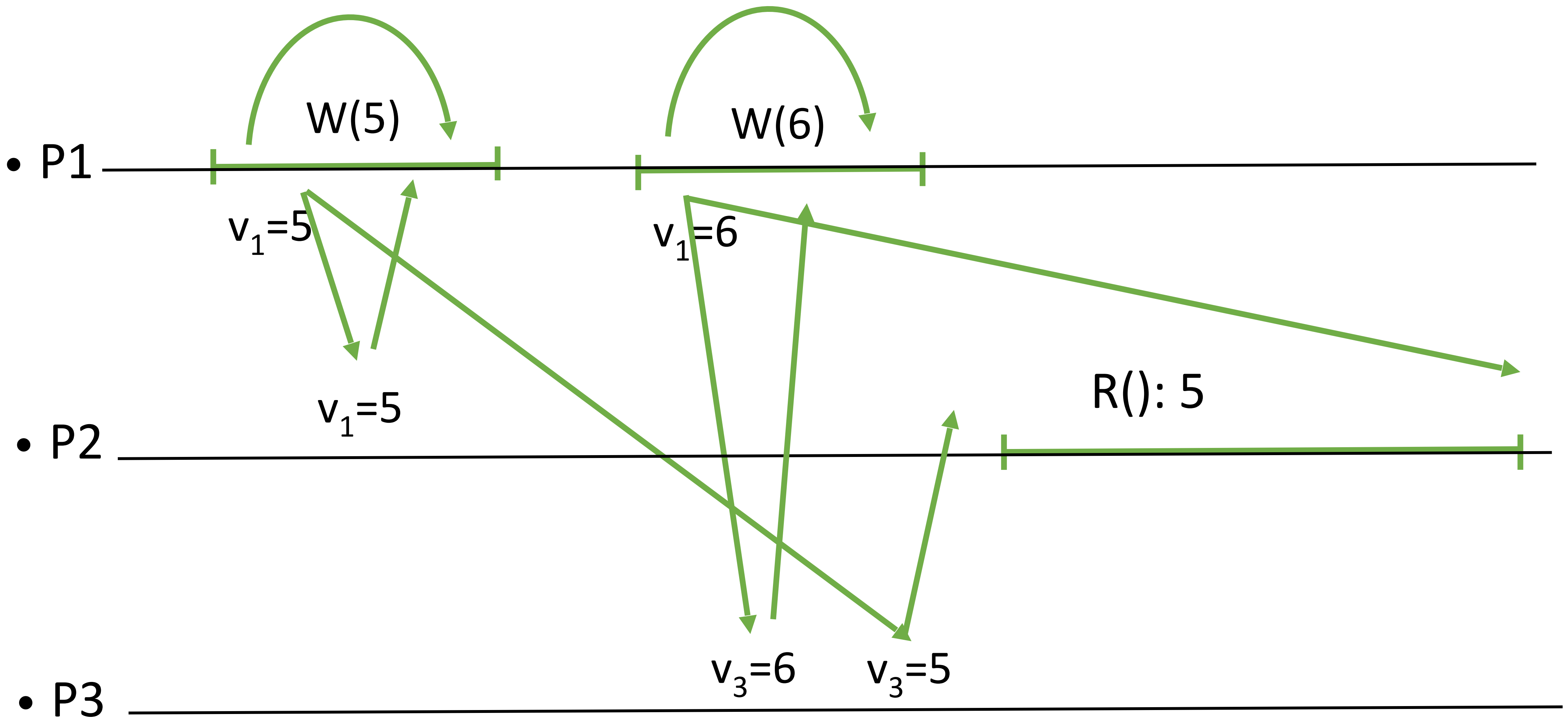
Incorrect execution. In p_3, the write message with sn_1 = 1 should be ignored.

Old Writes



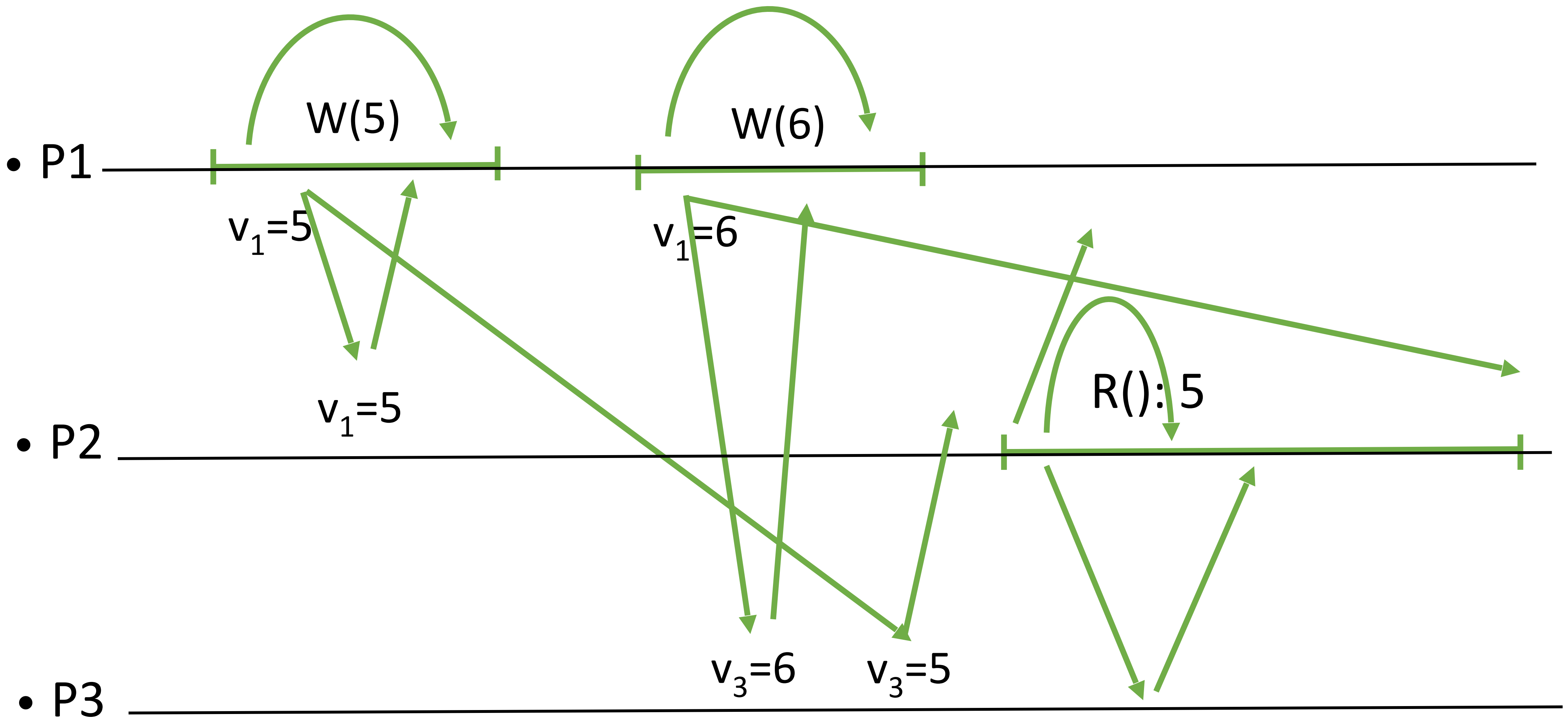
Incorrect execution. In p_3, the write message with sn_1 = 1 should be ignored.

Old Writes



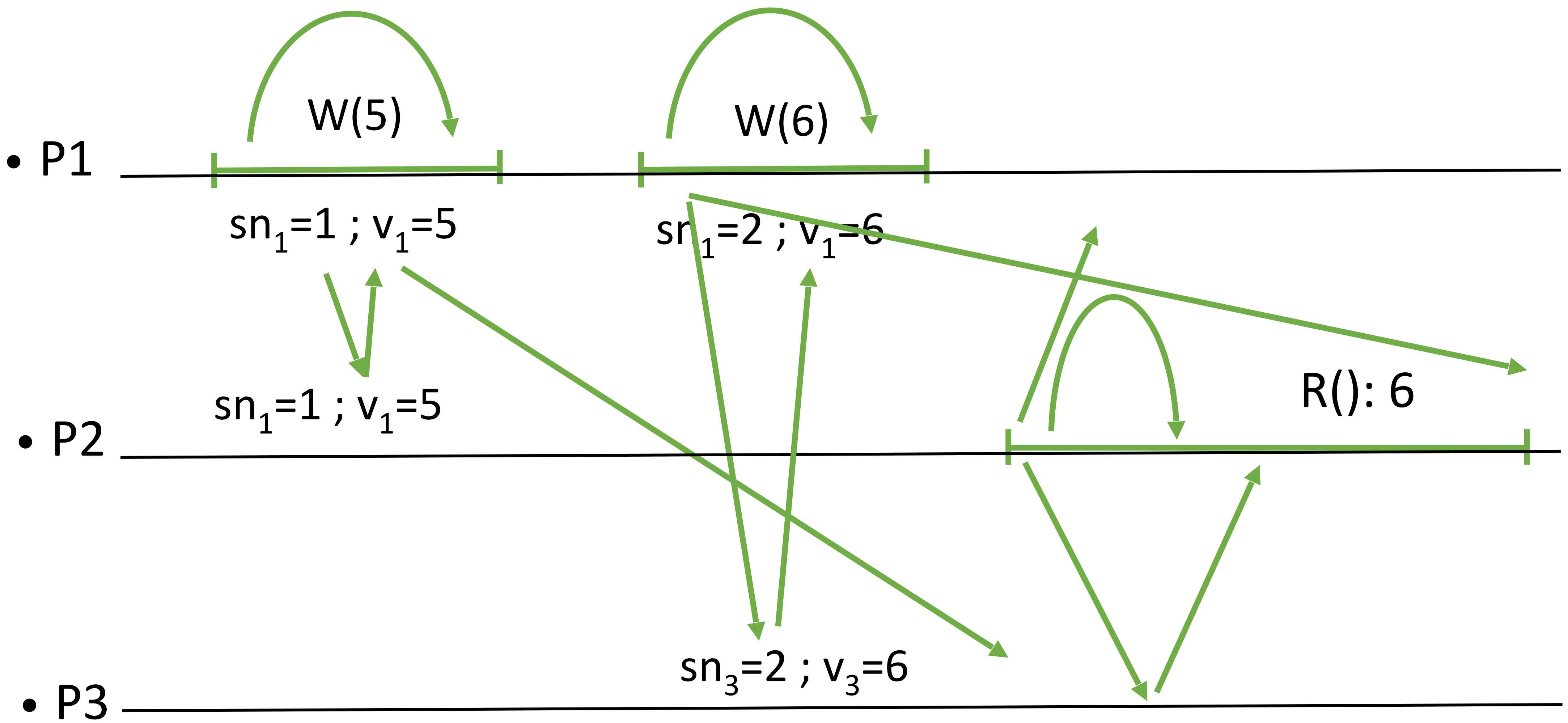
Incorrect execution. In p_3, the write message with sn_1 = 1 should be ignored.

Old Writes



Incorrect execution. In p_3, the write message with $sn_1 = 1$ should be ignored.

Old Writes



Correct execution. In p_3, the write message with $sn_1 = 1$ should be ignored.

Protocol - Write()

upon Write(v) at p_1

$ts_1 := ts_1 + 1$

trigger send [W,ts_1,v] to all

wait for deliver [W,ts_1,ack] from majority

trigger ok

At p_i

upon deliver [W,ts_1,v] from p_1

if $ts_1 > sn_i$ **then**

$v_i := v$

$sn_i := ts_1$

trigger send [W,ts_1,ack] to p_1

The timestamp ts_1 is sent with the ack messages to distinguish different writes.

The write messages that arrive late (with timestamps ts_1 less than sn_i) are ignored.

Protocol - Read()

upon Read() at pi

$rs_i := rs_i + 1$

trigger send $[R, rs_i]$ to all

wait for deliver $[R, rs_i, sn_j, v_j]$ from majority

$v := v_j$ with the largest sn_j

trigger Ret(v)

At p_i

upon deliver $[R, rs_j]$ from p_j

trigger send $[R, rs_j, sn_i, v_i]$ to p_j

The timestamp rs_i is used to distinguish different read requests from the process p_i.

The process p_i itself can be one of the processes in the quorum that replies with a value.

Correctness (liveness)

Every Read() or Write() eventually returns.

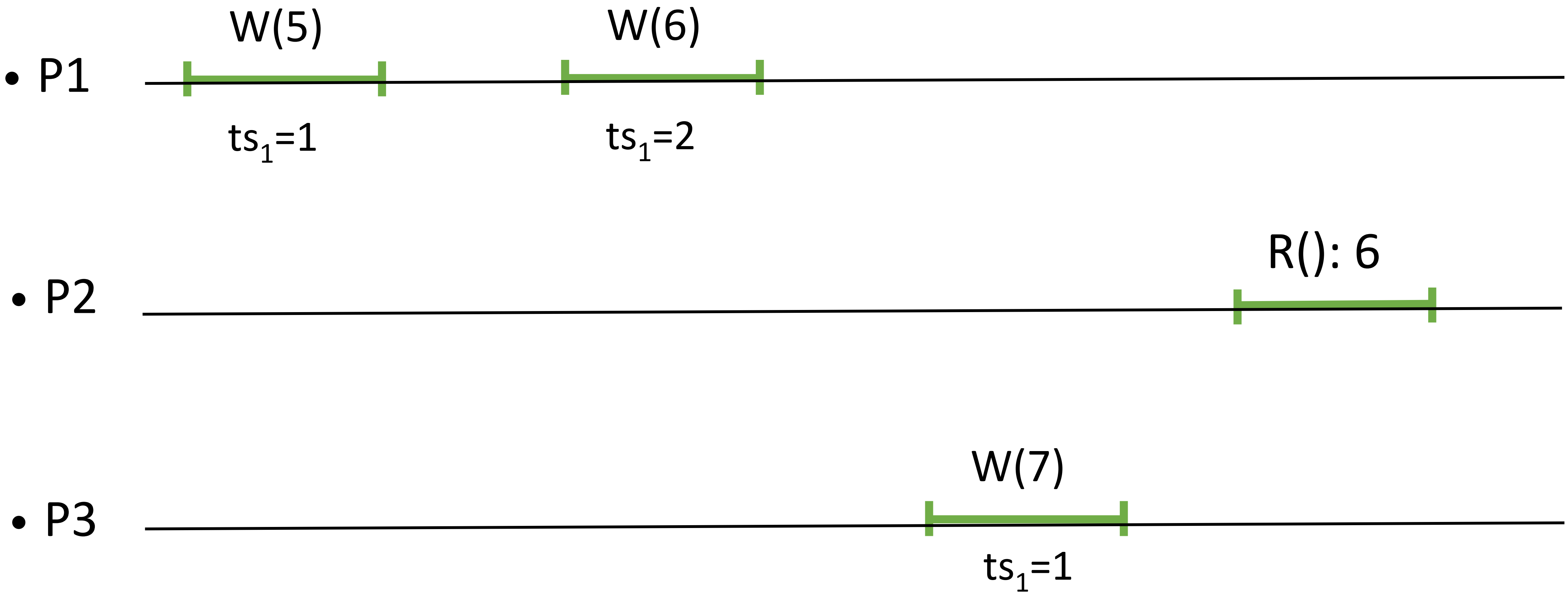
- As a majority of processes are correct, they will send the required number of acknowledgements.
- In the write case, a process may have a newer timestamp and may not send an ack. This means that a later write has written to it. Thus, a later Write() is started in p_1. Because writes execute in sequence in p_1, the older Write() has already returned.

Correctness (safety)

- 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 q_1 have x as their local value together with the highest timestamp in the system. Any subsequent Read() invocation by some process p_j reads values from a majority of processes q_2. The two quorums q_1 and q_2 intersect in at least one process p. Therefore, p_j can get the value x with the highest timestamp from p, and return x.

- A Read() returns the last value written or the value concurrently written.
  - Consider two writes w_1 and w_2 that execute and finish in sequence. The second one has a higher timestamp, and writes into a quorum q_2.
  - A value that a read returns is the value with the highest timestamp from a quorum q_3. The quorums q_2 and q_3 intersect at a process. Thus, w_2 does not miss the larger timestamp and the later value written by w_2.

Multiple writers



Incorrect execution. The protocol does not support multiple writers. The write in p_3 is ignored and results in the incorrect read in p_2. We will see an N-N atomic register in the next lectures.

Original slides adopted from R. Guerraoui