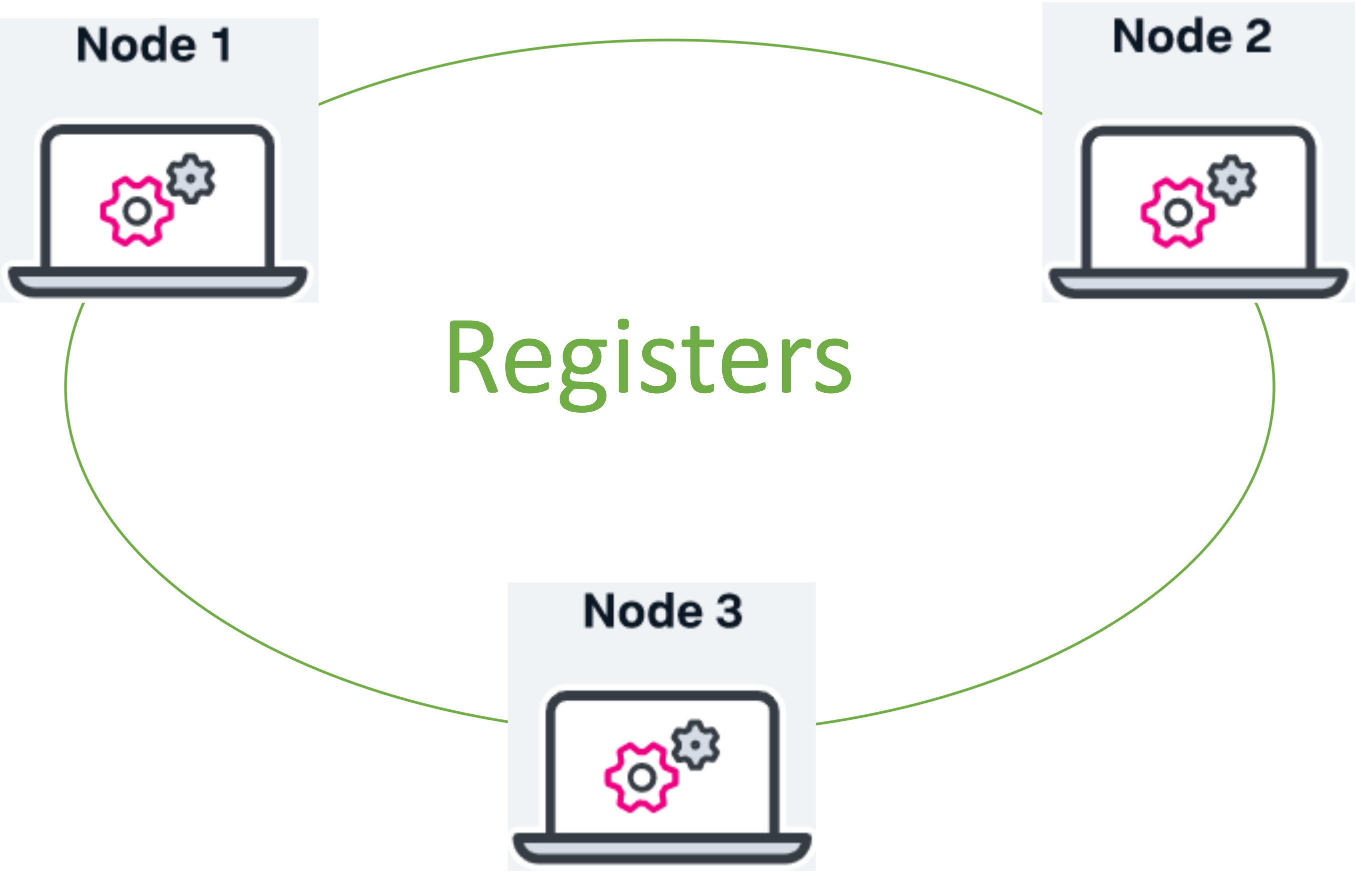
Atomic registers

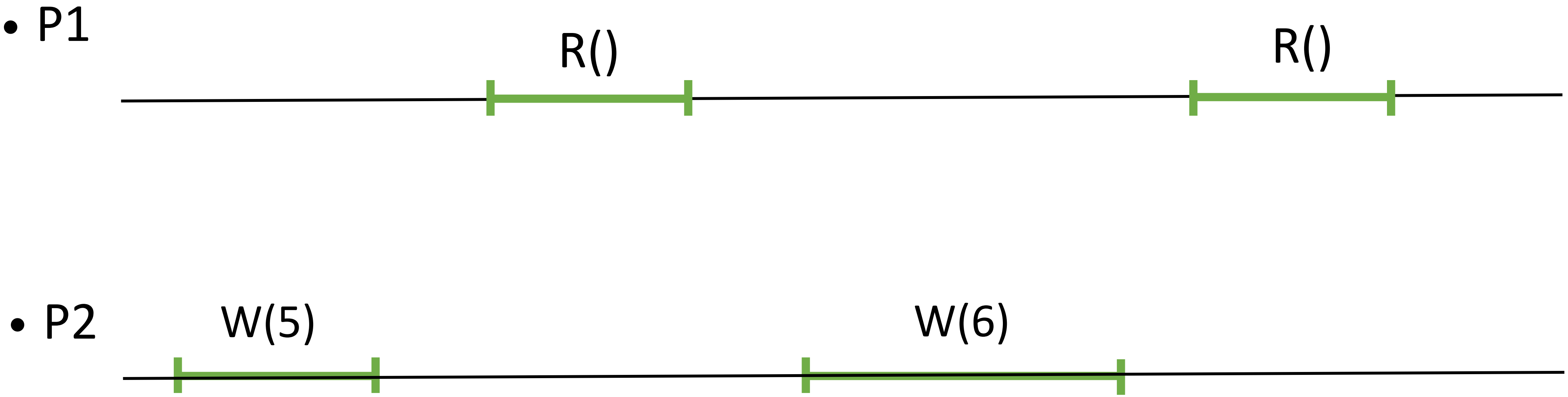
Mohsen Lesani

Atomic register specification

The application model



Sequential execution



- P1

R()

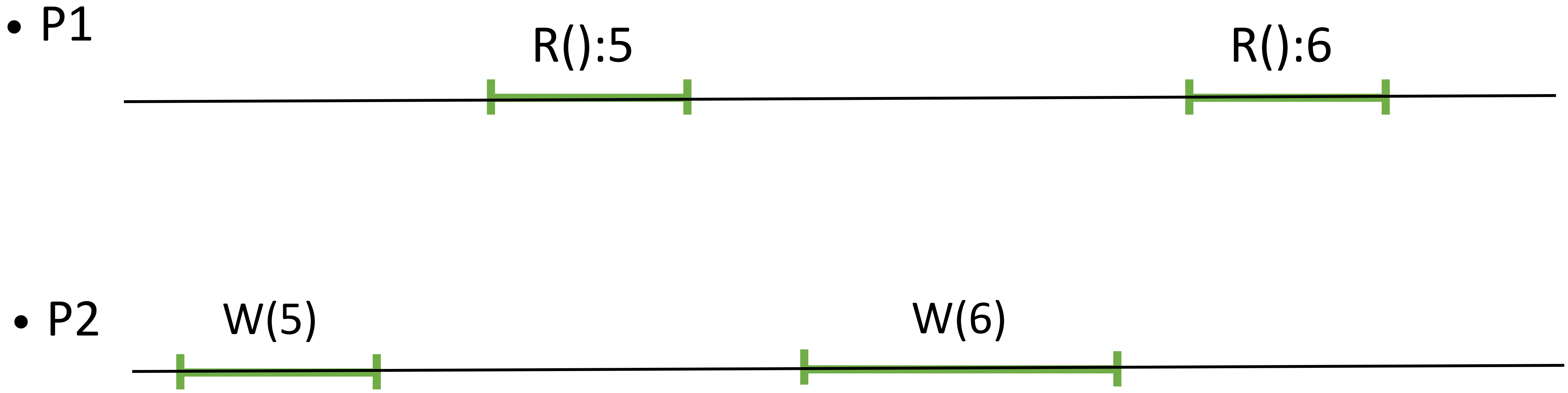
R()

- P2

W(5)

W(6)

Sequential execution



- P1

R():5

R():6

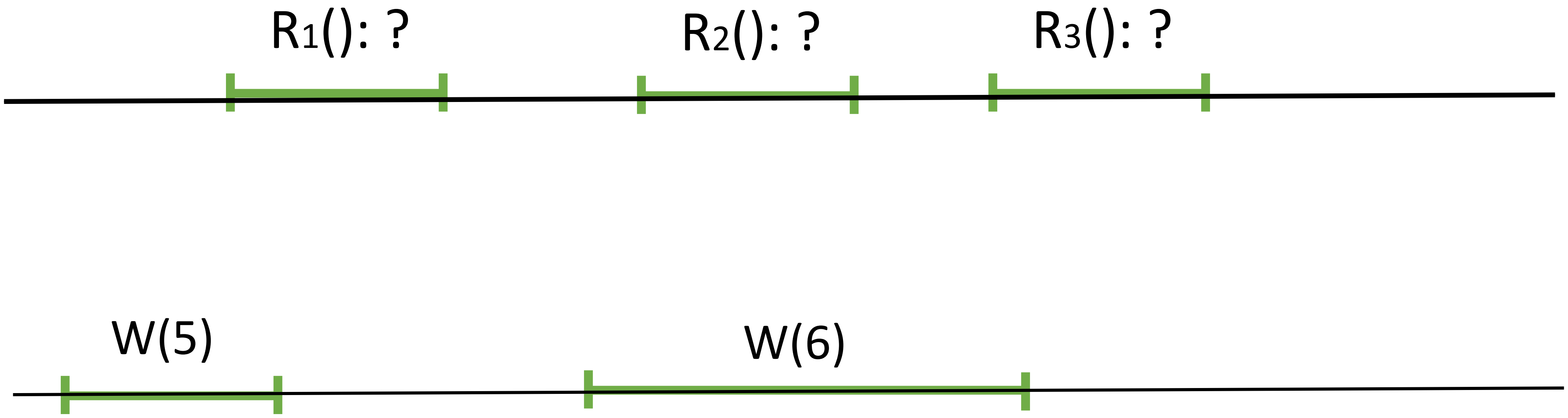
- P2

W(5)

W(6)

Concurrent execution

- P1
- P2



R_1(): ?

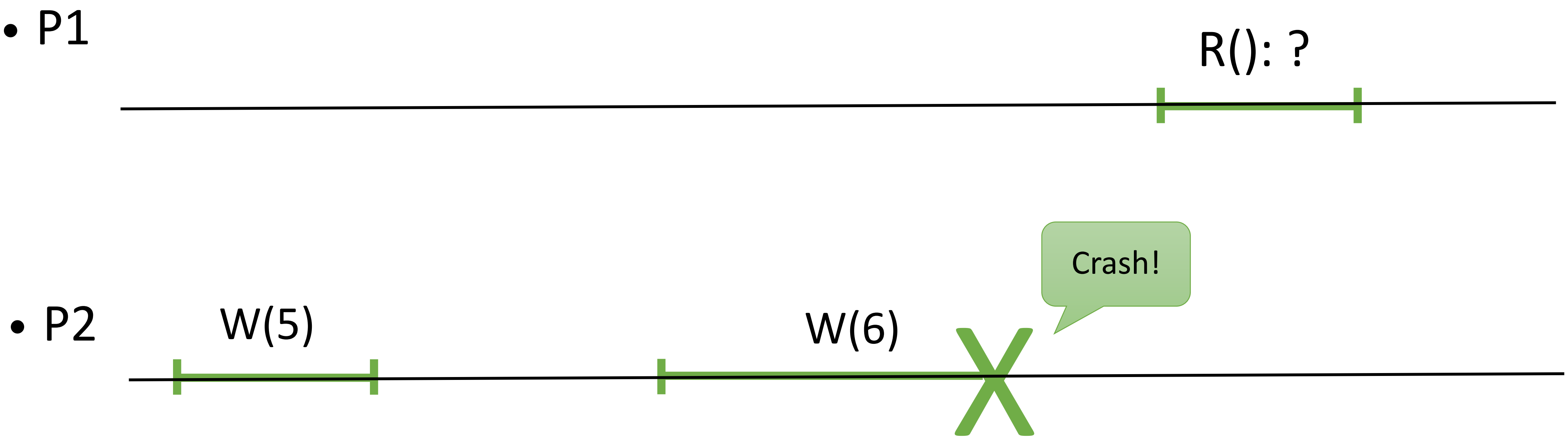
R_2(): ?

R_3(): ?

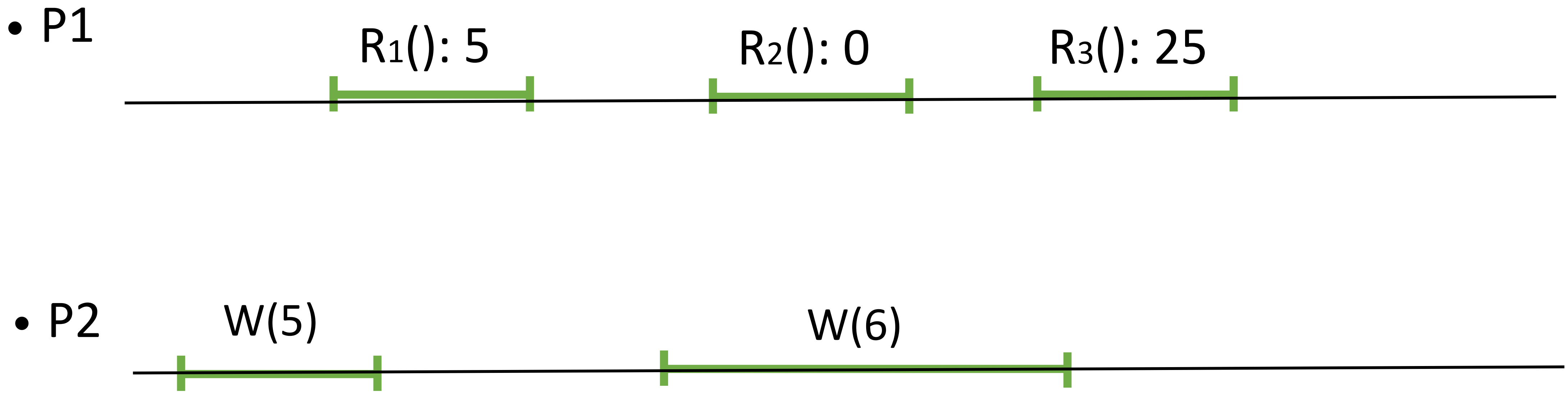
W(5)

W(6)

Execution with failures

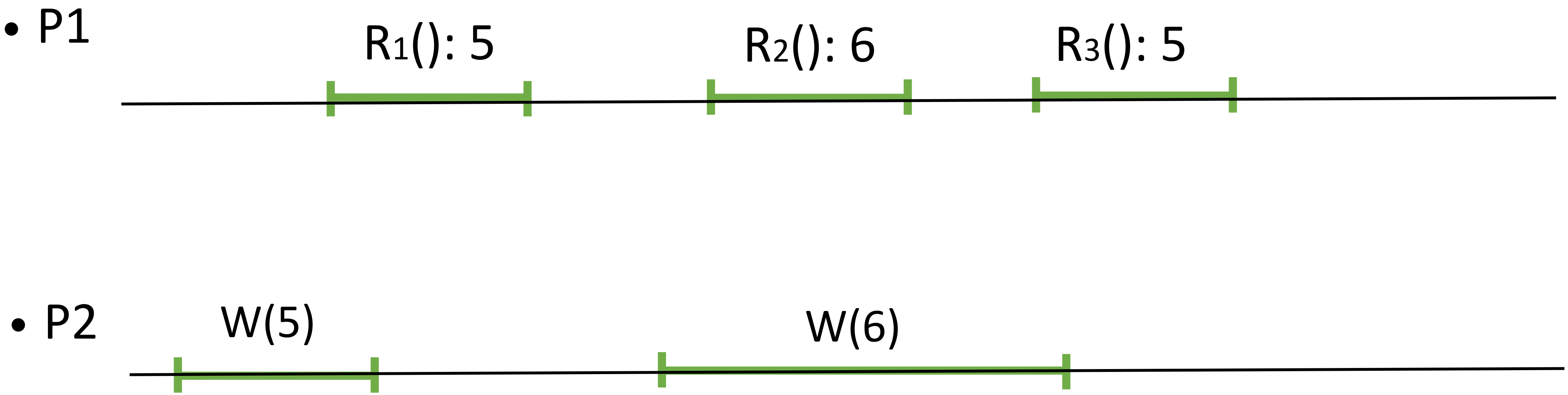


Execution 1



Just a so-called safe execution. Not a regular execution. Not an atomic execution.
R2 does not return the value of a previous or concurrent write.
No matter where W(6) is linearized, the return value of R2 cannot be justified.

Execution 2



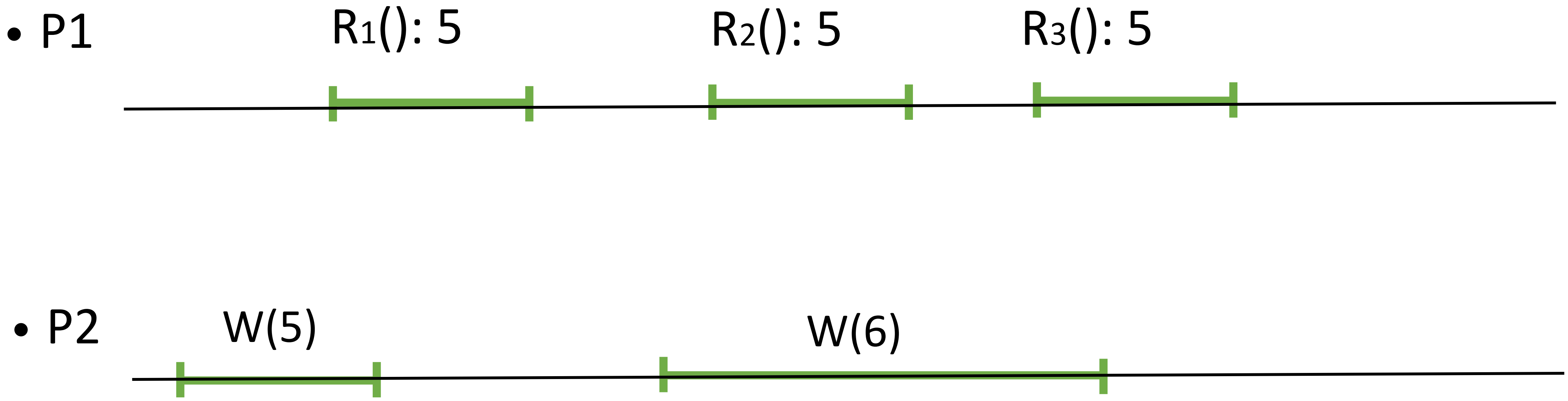
A regular execution. Not an atomic execution.
R2 returns the value of the concurrent write W(6). R3 returns the value of the lates write W(5).
W(6) can be linearized before R2 to justify its return value. However, the return value of R3 cannot be justified.

Regular vs Atomic

- 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.

- The atomic register does not allow that.

Execution 3



- P1

R_1(): 5

R_2(): 5

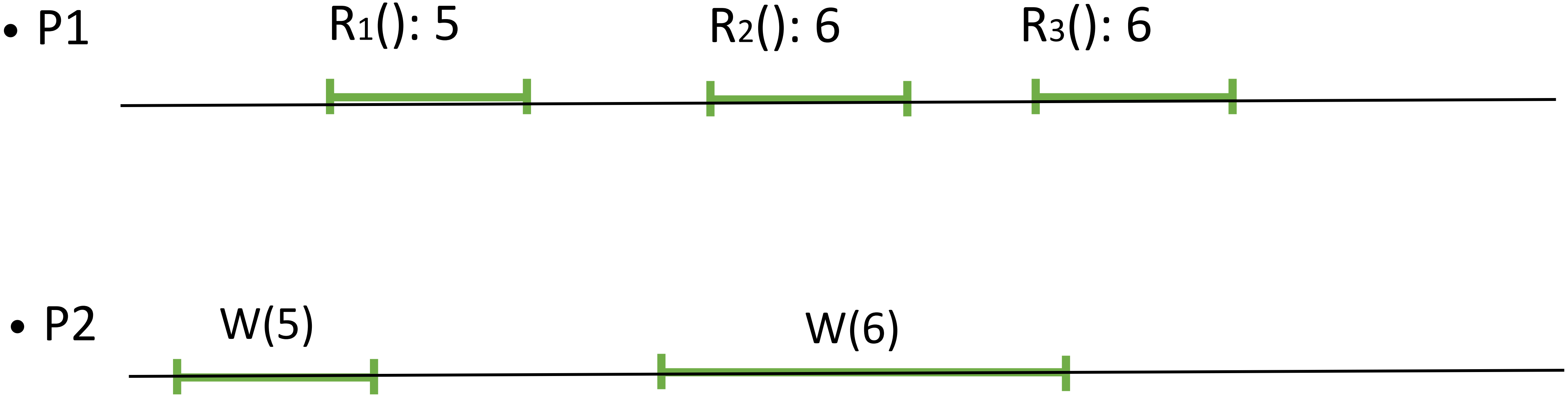
R_3(): 5

- P2

W(5)

W(6)

Execution 4



- P1

R_1(): 5

R_2(): 6

R_3(): 6

- P2

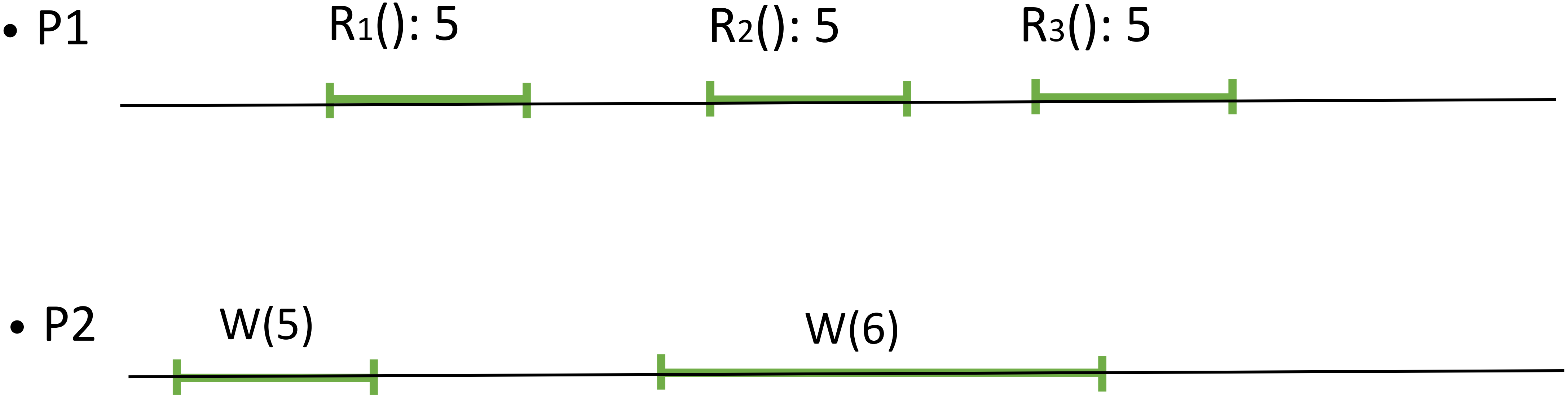
W(5)

W(6)

Safety

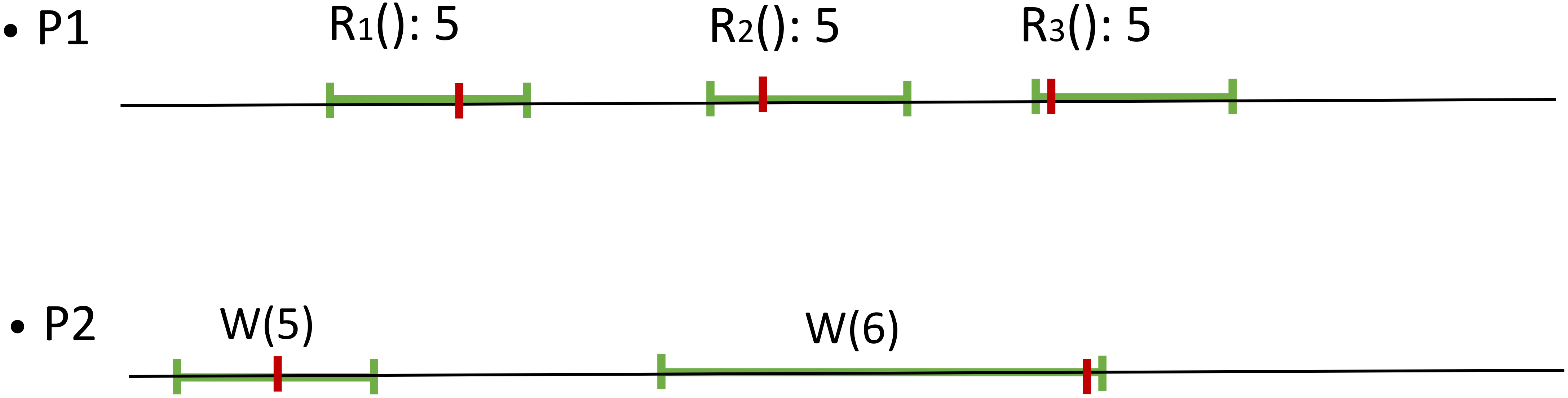
- **An atomic register** provides strong guarantees even when there is concurrency and failures
- Every operation appears to be executed at some instant between its invocation and response events.
- The execution is equivalent to a sequential and failure-free execution (called the **linearization**).

Execution 3



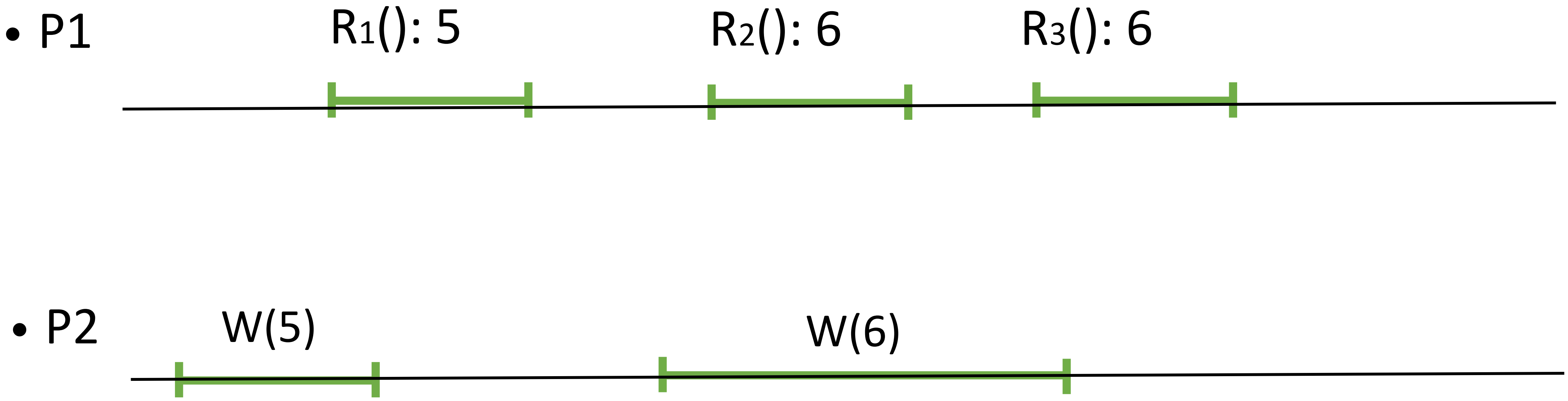
An atomic execution. W(6) can be linearized after both R2 and R3. And the return value of both can be justified.

Execution 3



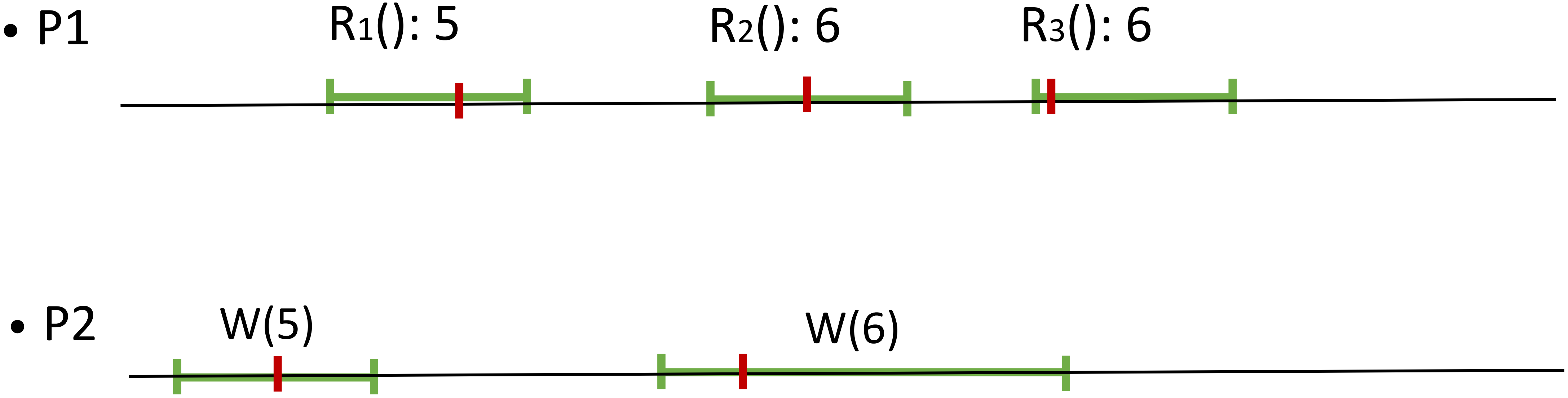
An atomic execution. W(6) can be linearized after both R2 and R3. And the return value of both can be justified.

Execution 4



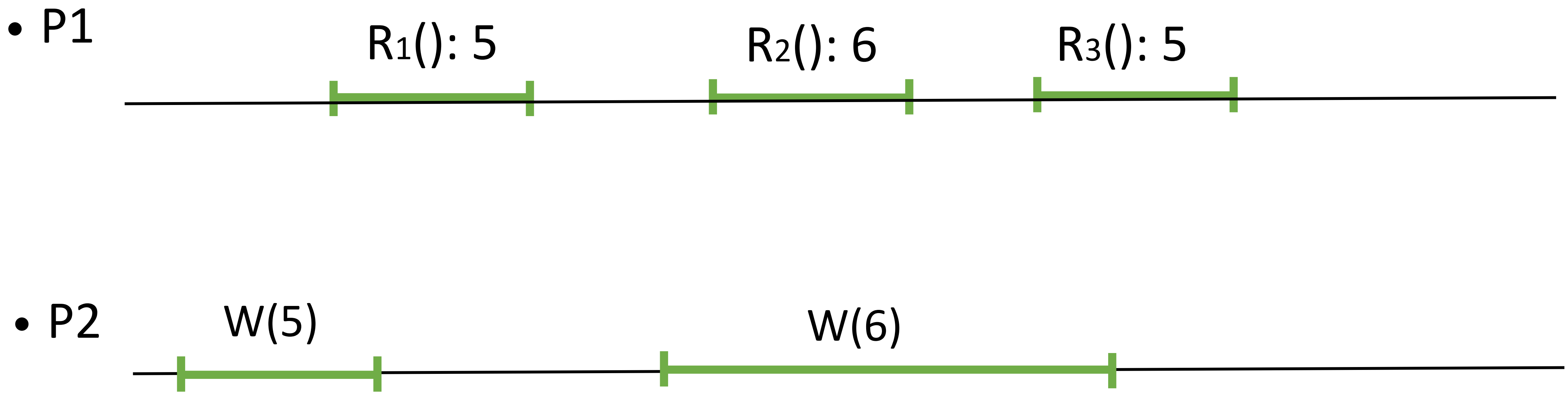
An atomic execution. W(6) can be linearized before both R2 and R3. And the return value of both can be justified.

Execution 4



An atomic execution. W(6) can be linearized before both R2 and R3. And the return value of both can be justified.

Revisit Execution 2

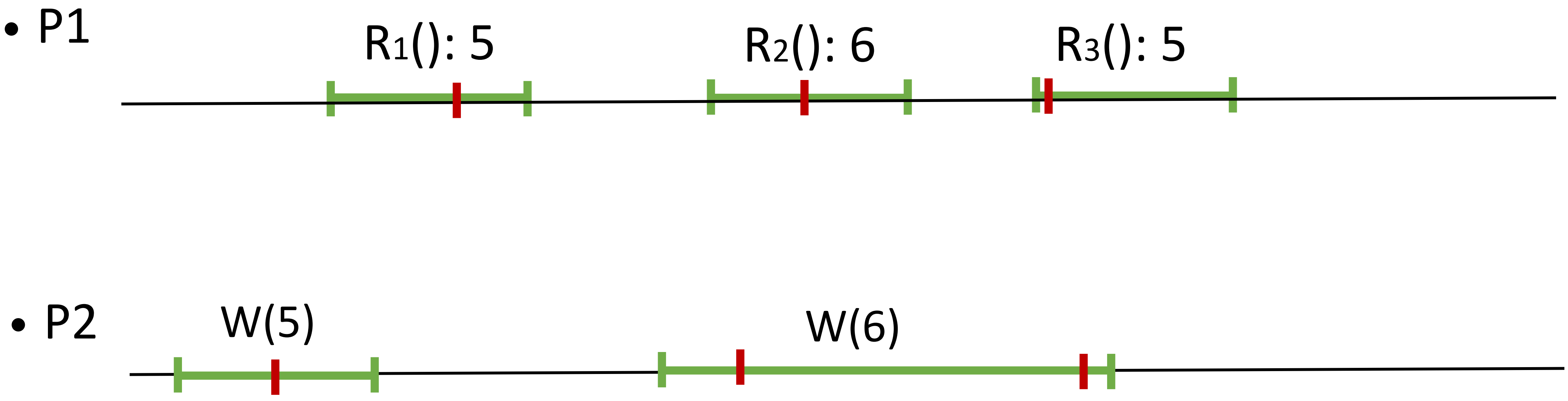


A regular execution. Not an atomic execution.

R2 returns the value of the concurrent write W(6). R3 returns the value of the lates write W(5).

W(6) can be linearized before R2 to justify its return value. However, the return value of R3 cannot be justified.

Revisit Execution 2

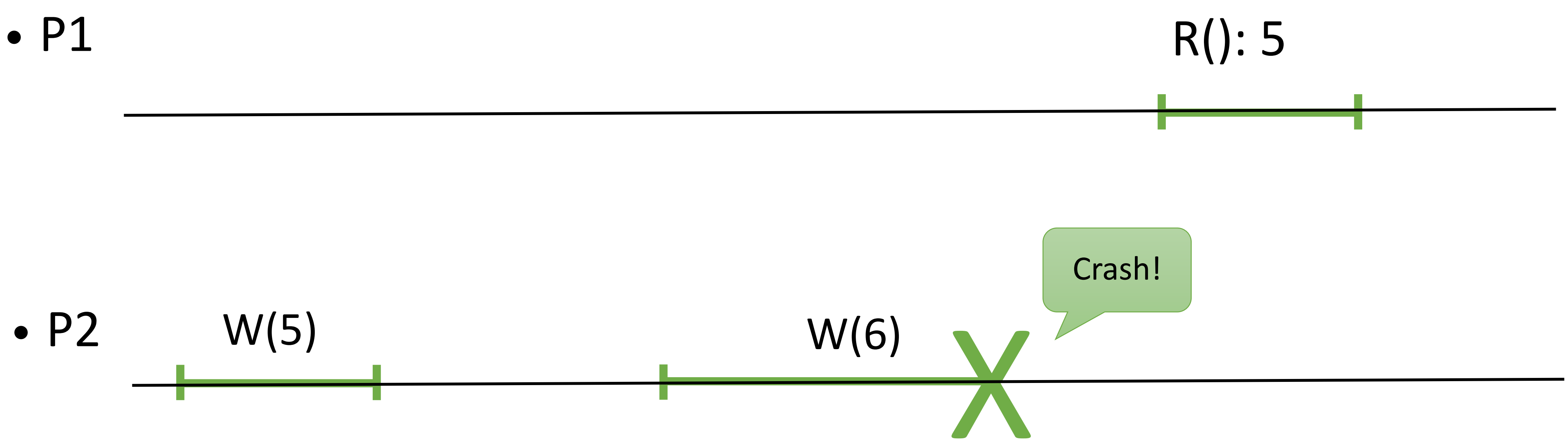


A regular execution. Not an atomic execution.
R2 returns the value of the concurrent write W(6). R3 returns the value of the lates write W(5).
W(6) can be linearized before R2 to justify its return value. However, the return value of R3 cannot be justified.

Atomic register

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

Execution 5



- P1

R(): 5

- P2

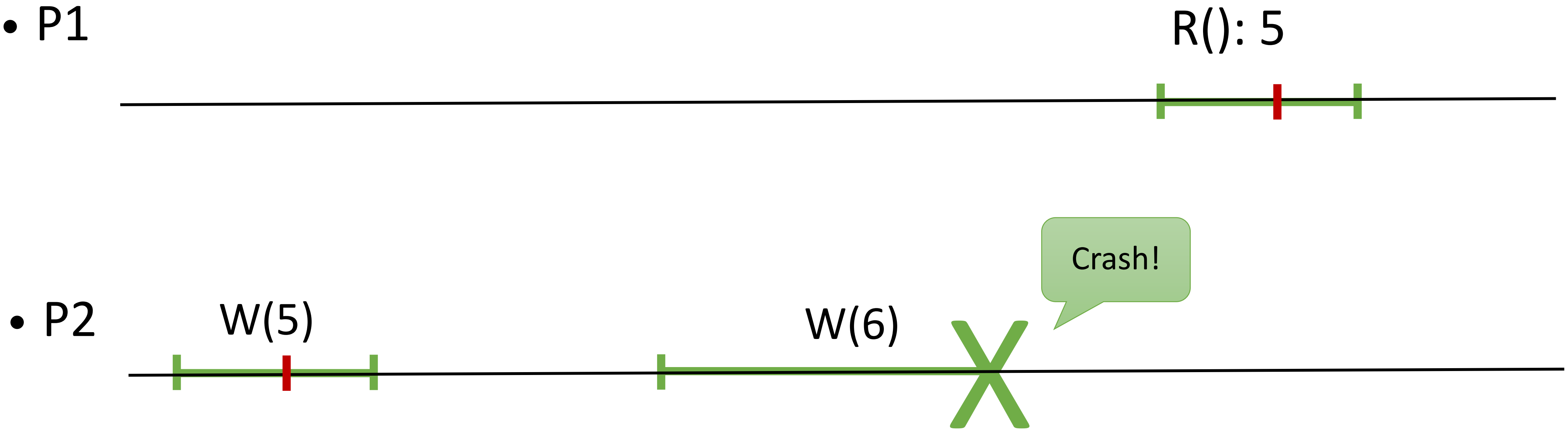
W(5)

W(6)

Crash!

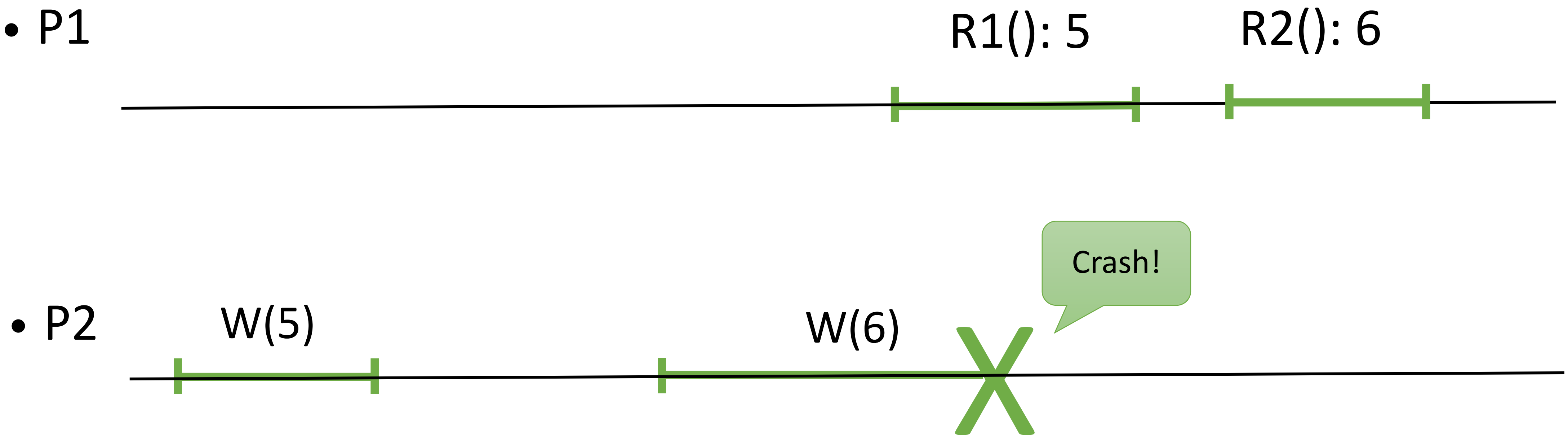
An atomic execution. W(6) is considered as not executed at all.

Execution 5



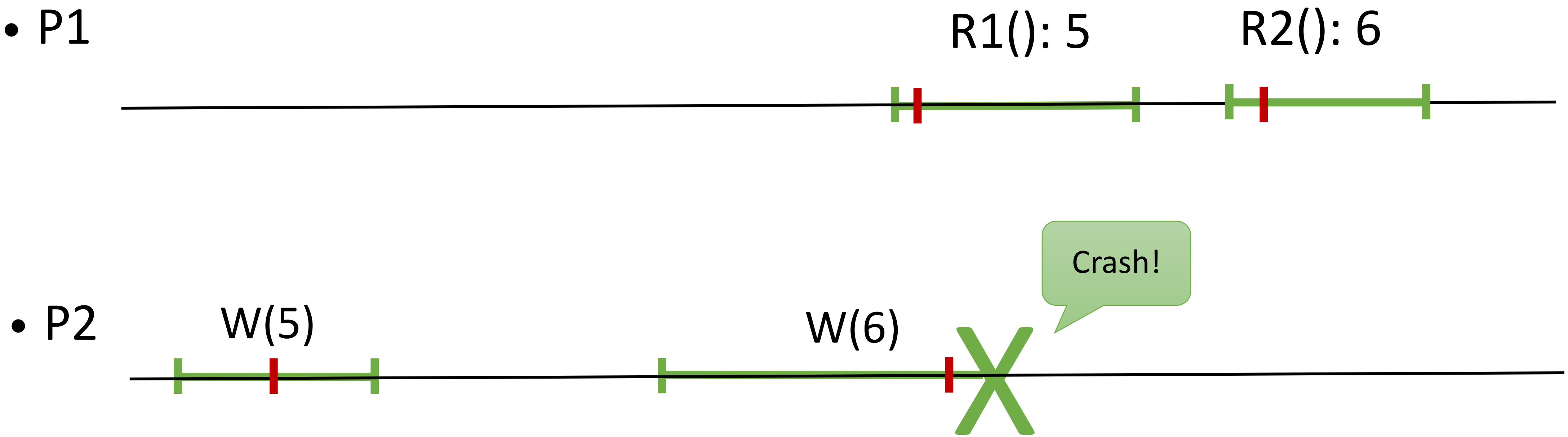
An atomic execution. W(6) is considered as not executed at all.

Execution 6



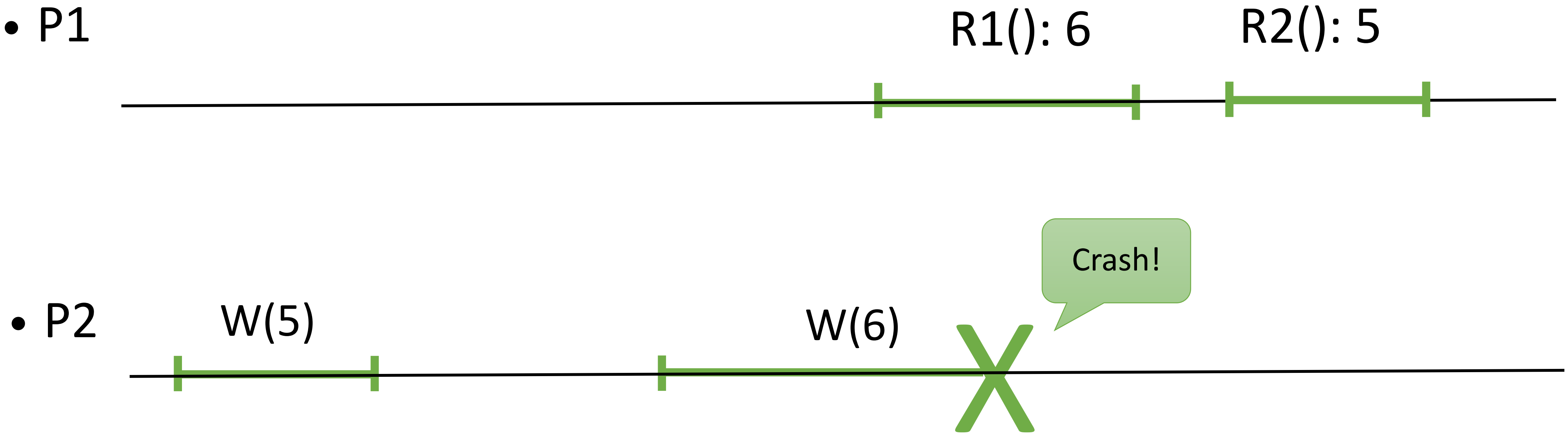
An atomic execution. W(6) can be linearized after R1 and before R2. And the return value of both can be justified.

Execution 6



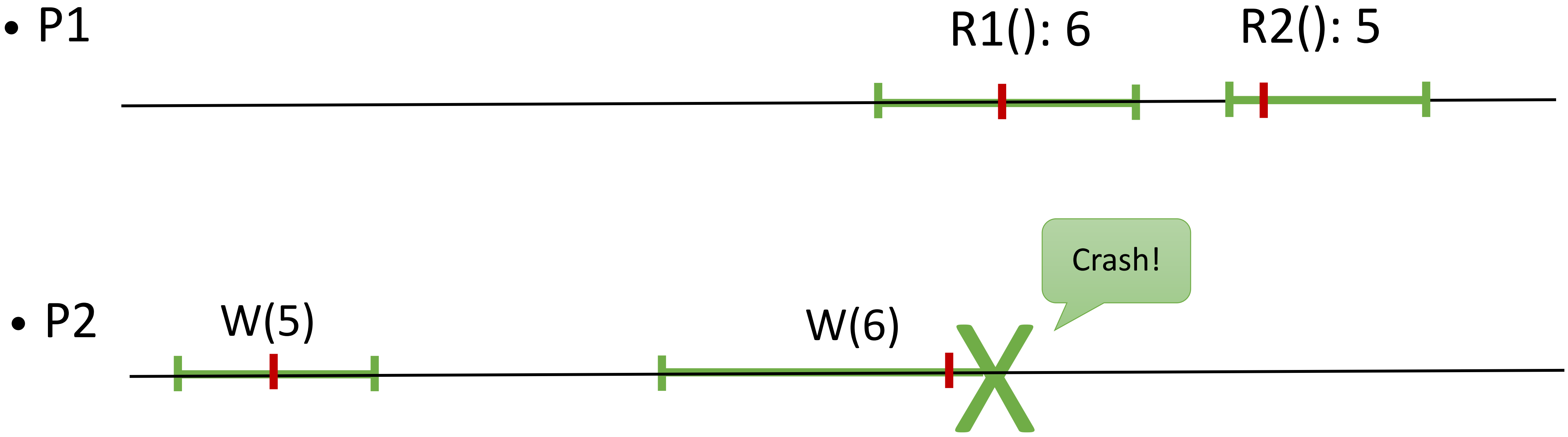
An atomic execution. W(6) can be linearized after R1 and before R2. And the return value of both can be justified.

Execution 7



A regular execution. Not an atomic execution.
R1 is returning the value of the concurrent write W(6). R2 is returning the value of the latest write W(5).
W(6) can be linearized before R1 to justify the return value of R1 but then the return value of R2 cannot be justified.

Execution 7



A regular execution. Not an atomic execution.
R1 is returning the value of the concurrent write W(6). R2 is returning the value of the latest write W(5).
W(6) can be linearized before R1 to justify the return value of R1 but then the return value of R2 cannot be justified.

Atomic register Algorithms

Overview of this lecture

1. **A 1-1 atomic fail-stop algorithm**
2. From regular to atomic
3. A 1-N atomic fail-stop algorithm
4. A N-N atomic fail-stop algorithm
5. From fail-stop to fail-silent

Fail-stop algorithms

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

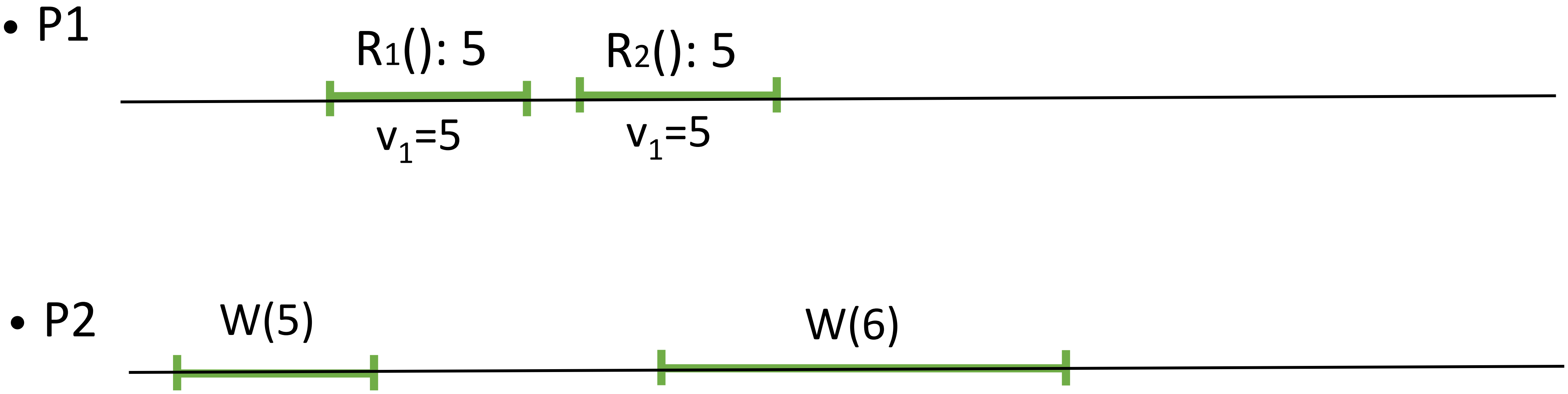
A fail-stop 1-1 atomic algorithm

```
upon Write(v) at $p_1$
    send [W,v] to $p_2$
    wait until either:
        deliver [ack] from $p_2$
        suspect [$p_2$]
    trigger ok
```

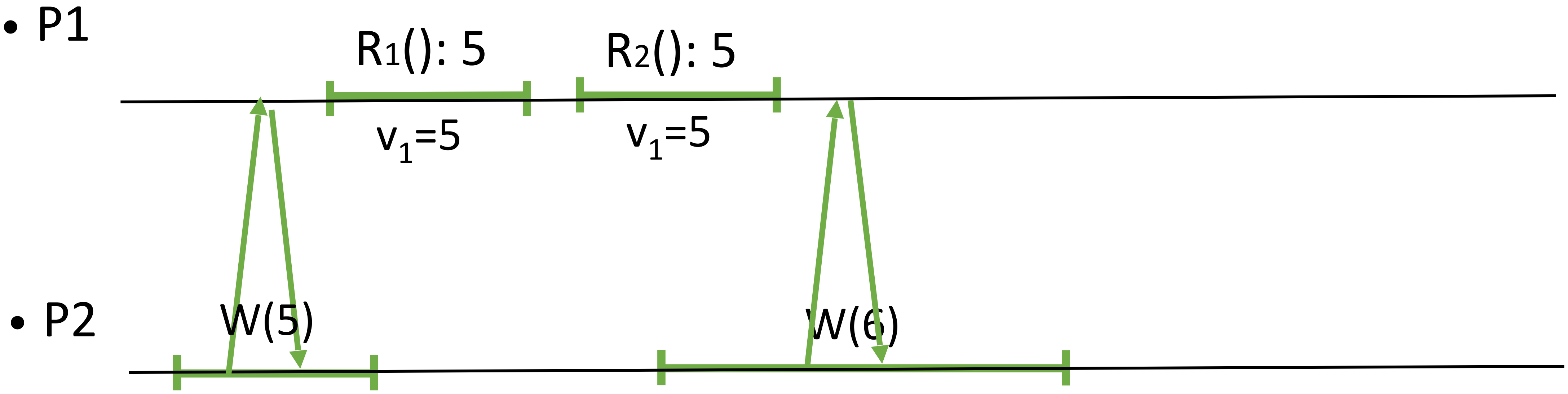
```
At $p_2$ :
    upon receive [W,v] from $p_1$
        $v_2$ := v
        trigger send [ack] to $p_2$

upon Read() at $p_2$
    trigger Ret($v_2$)
```

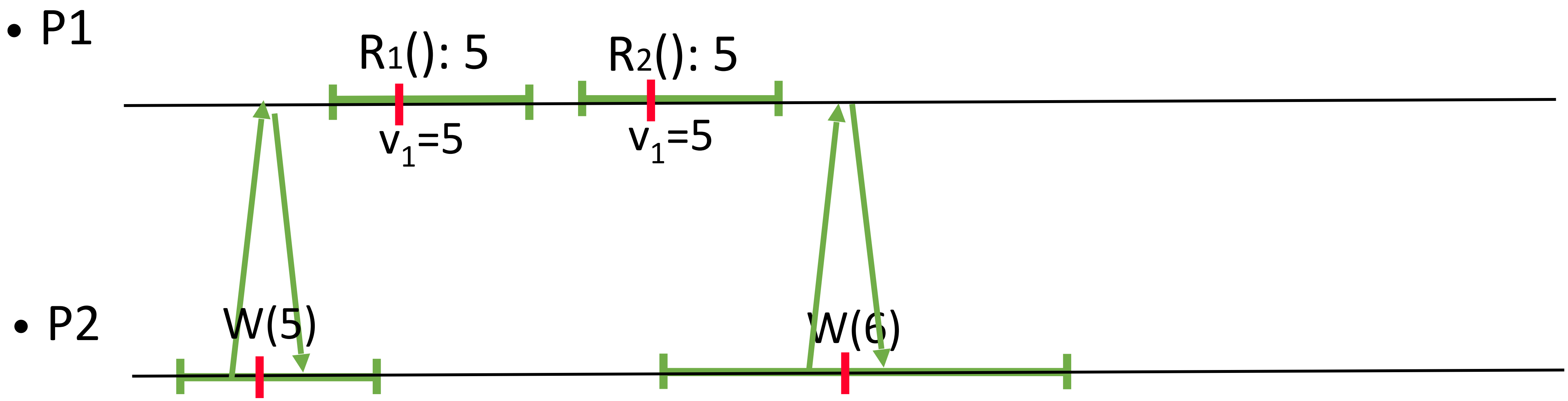
Atomicity?



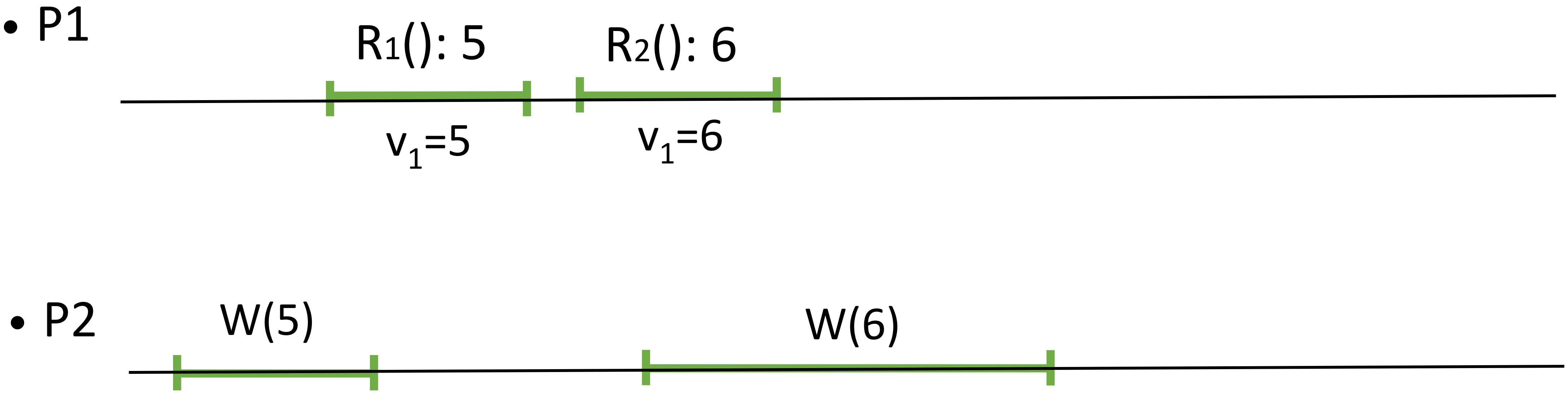
Atomicity?



Atomicity?

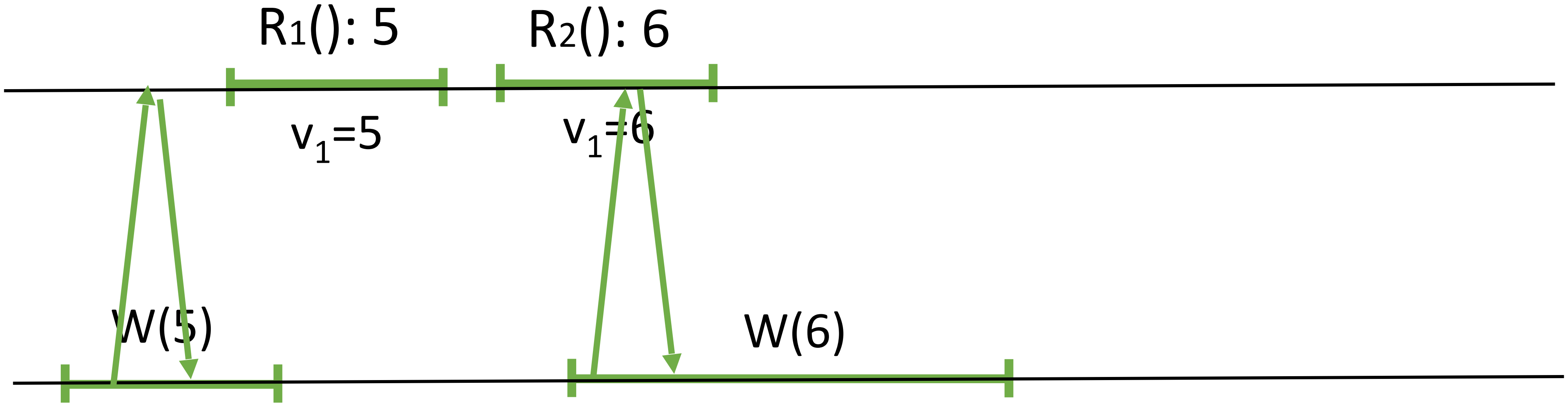


Atomicity?

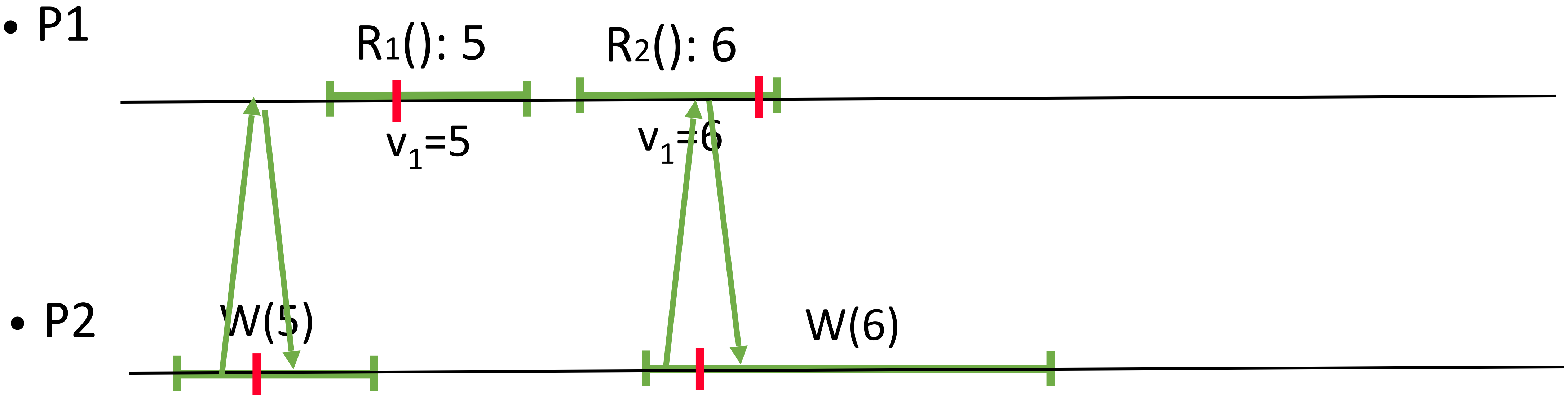


Atomicity?

- P1
- P2



Atomicity?



Overview of this lecture

1. A 1-1 atomic fail-stop algorithm
2. **From regular to atomic**
3. A 1-N atomic fail-stop algorithm
4. A N-N atomic fail-stop algorithm
5. From fail-stop to fail-silent

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

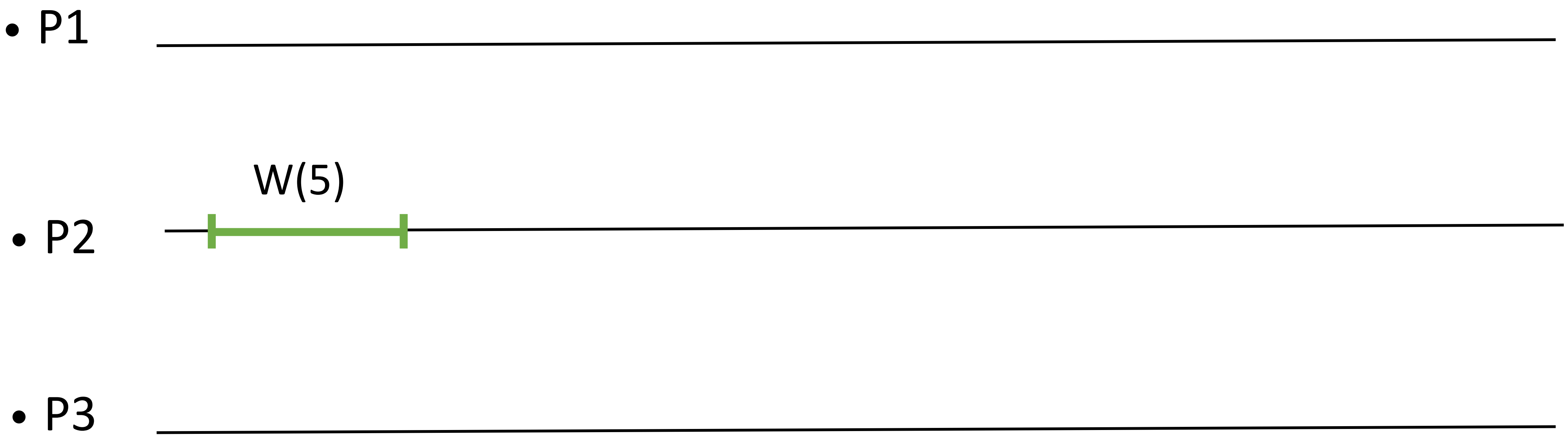
The regular algorithm

```
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 receive [W,v] from $p_j$
        $v_i$ := v
        trigger send [ack] to $p_j$

Read() at $p_i$
    trigger Ret($v_i$)
```

Atomicity?

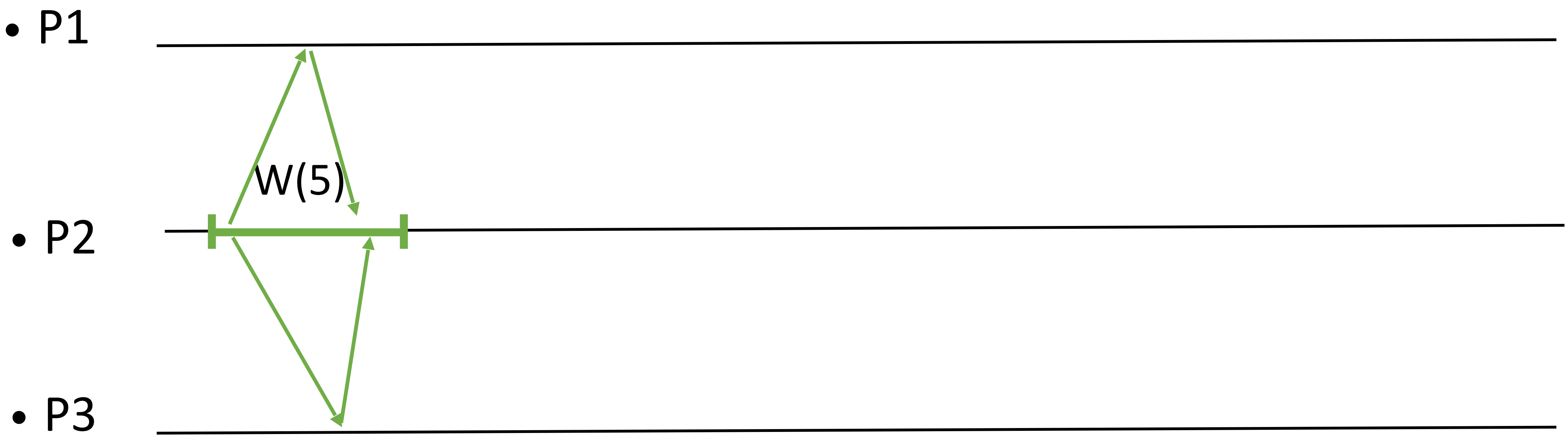


- P1

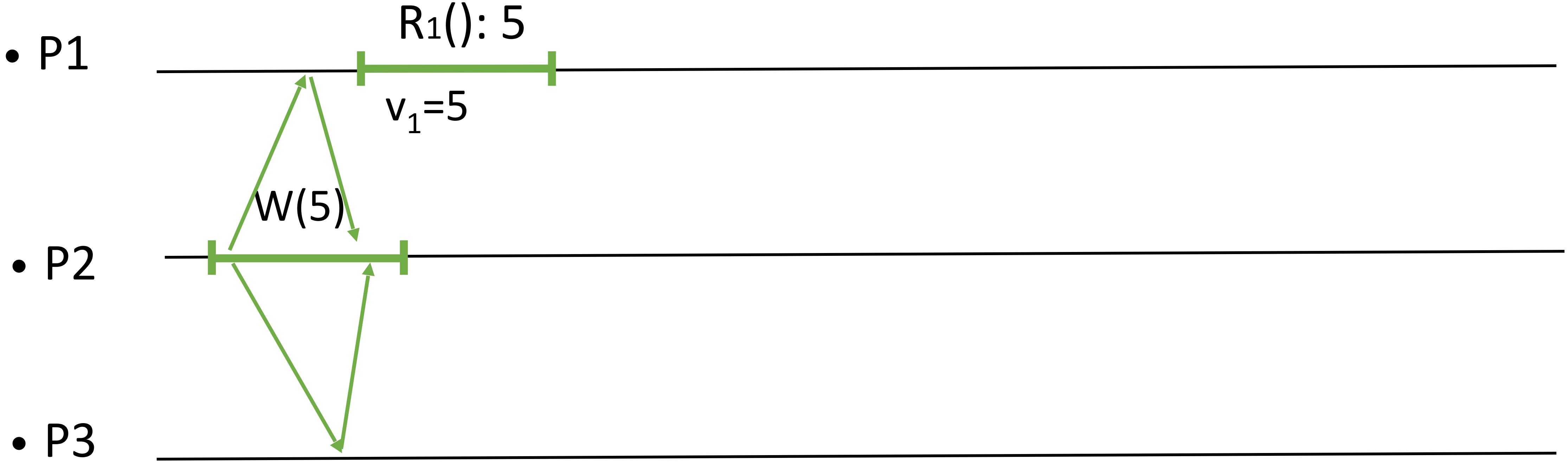
W(5)

- P2
- P3

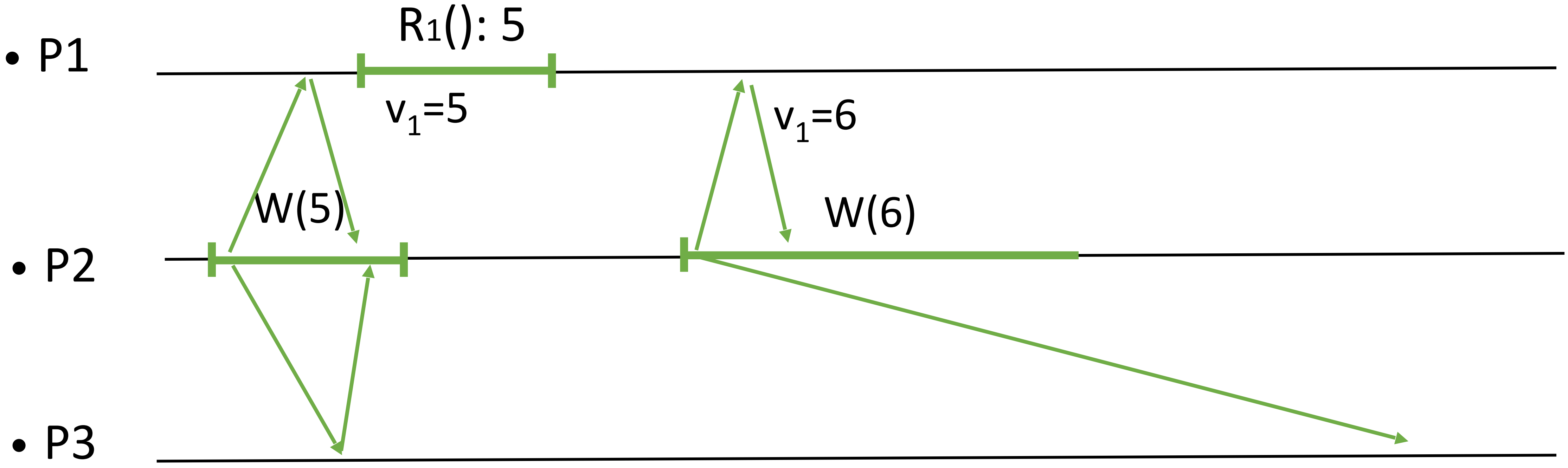
Atomicity?



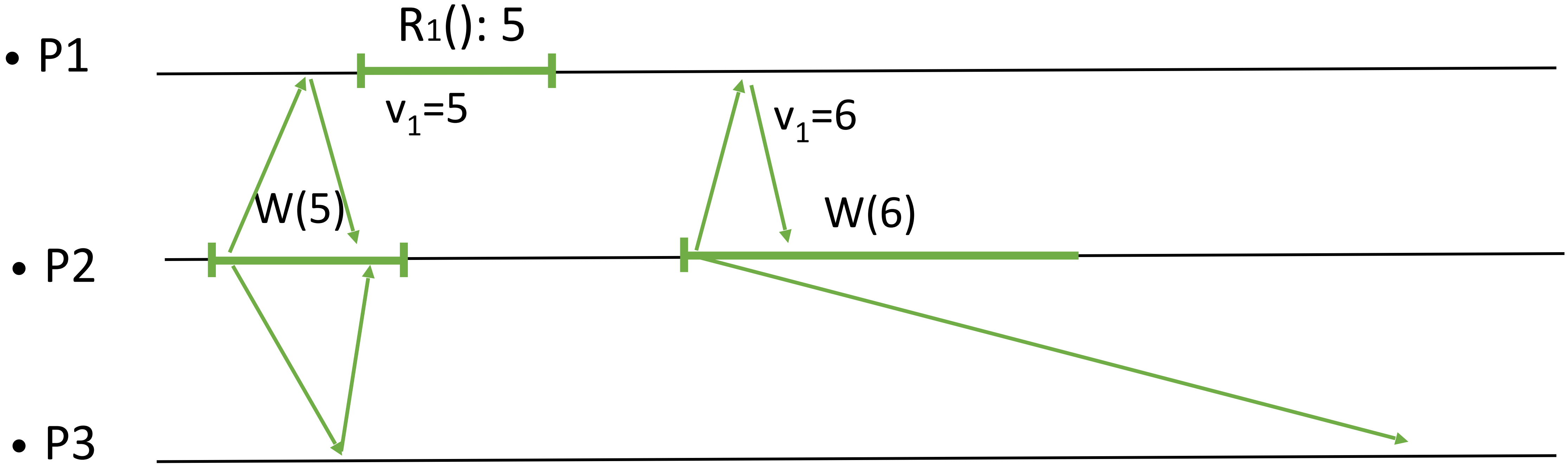
Atomicity?



Atomicity?

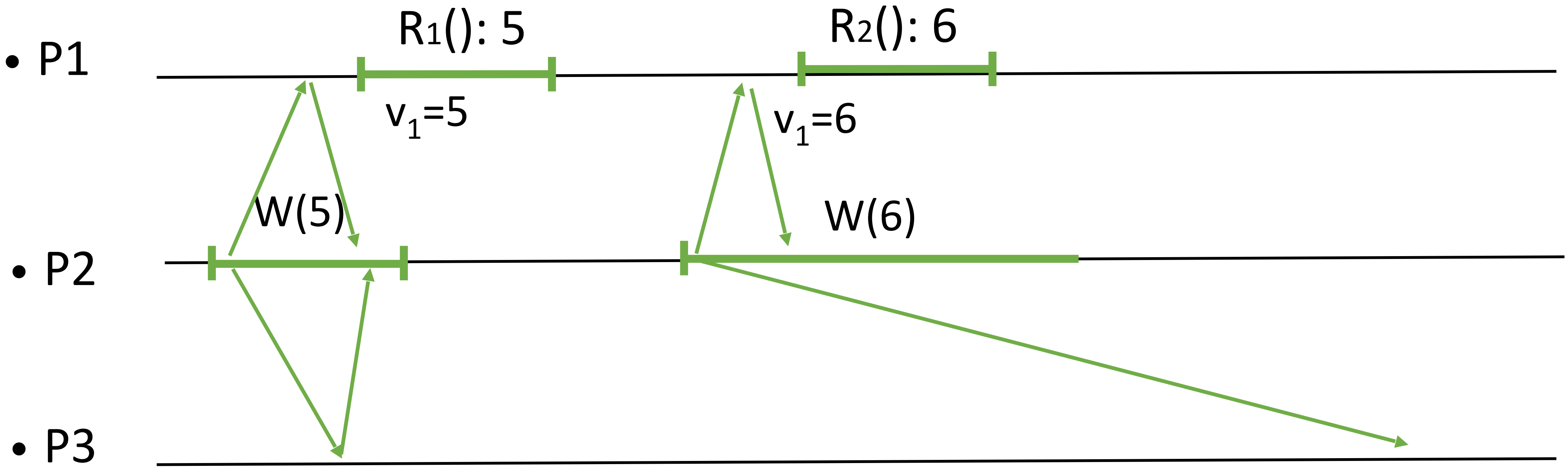


Atomicity?



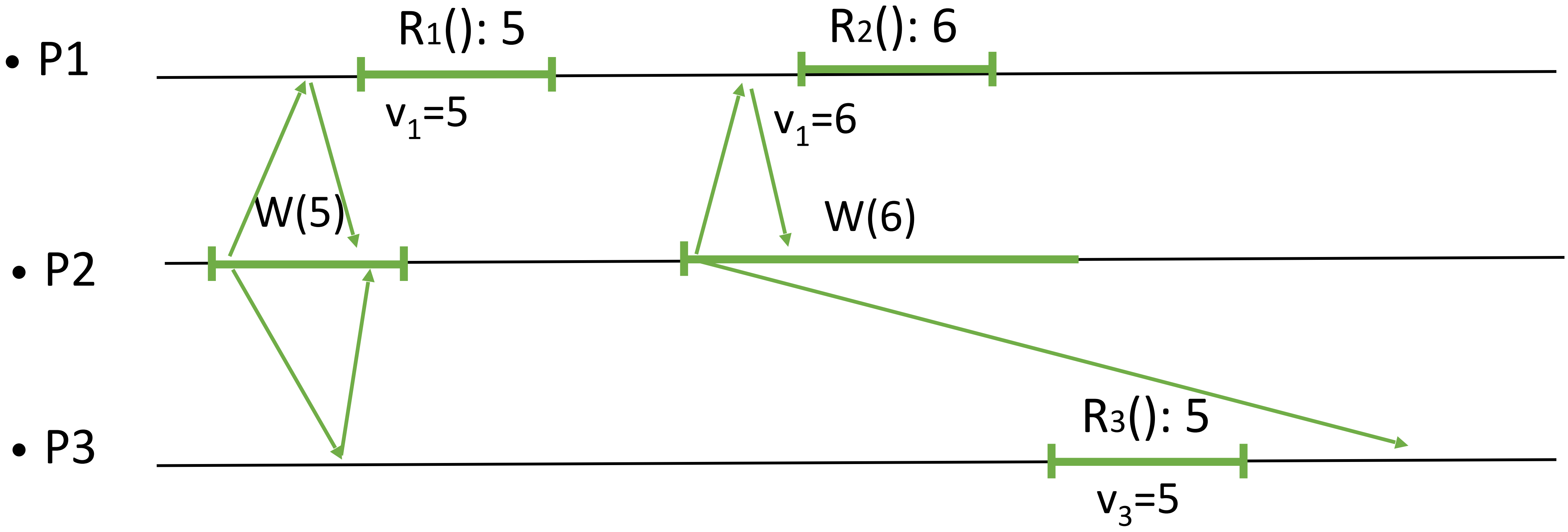
W(6) has updated P1 but not P3 yet.

Atomicity?



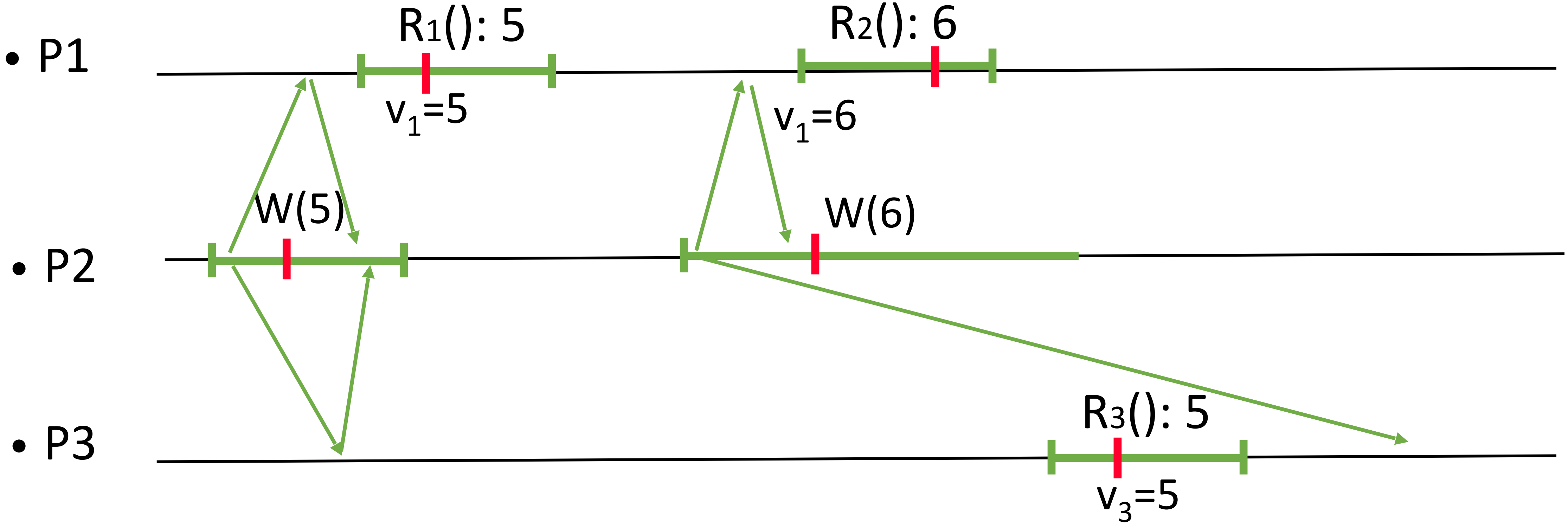
W(6) has updated P1 but not P3 yet.

Atomicity?



W(6) has updated P1 but not P3 yet.

Atomicity?



W(6) has updated P1 but not P3 yet.

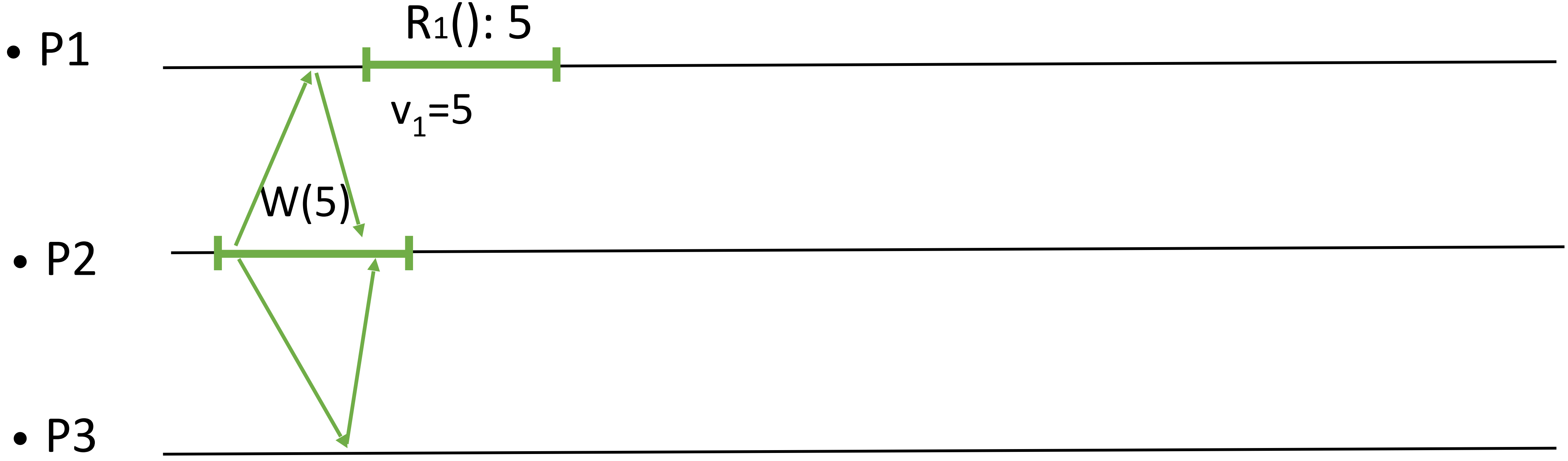
R3 should return 6.

Fix? Reads write.

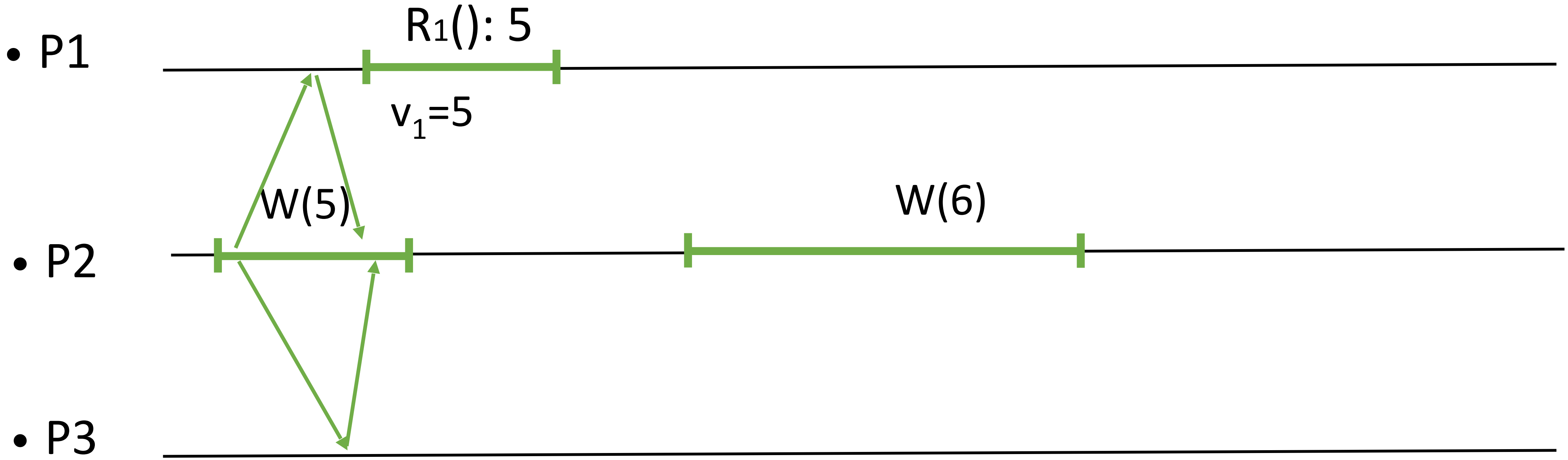
```
upon Read() at $p_i$
    trigger send [W,$v_i$] to all
    foreach $p_j$, wait until either:
        deliver [ack] or
        suspect [$p_j$]
    trigger Ret($v_i$)
```

Reads update the other processes before returning the value.

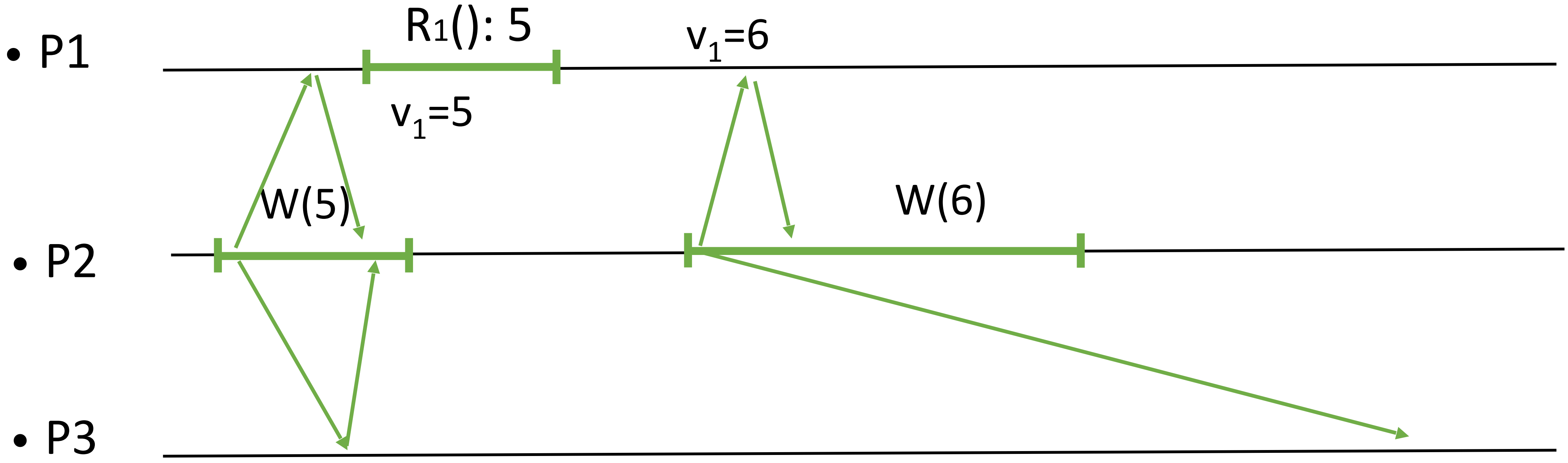
Atomicity?



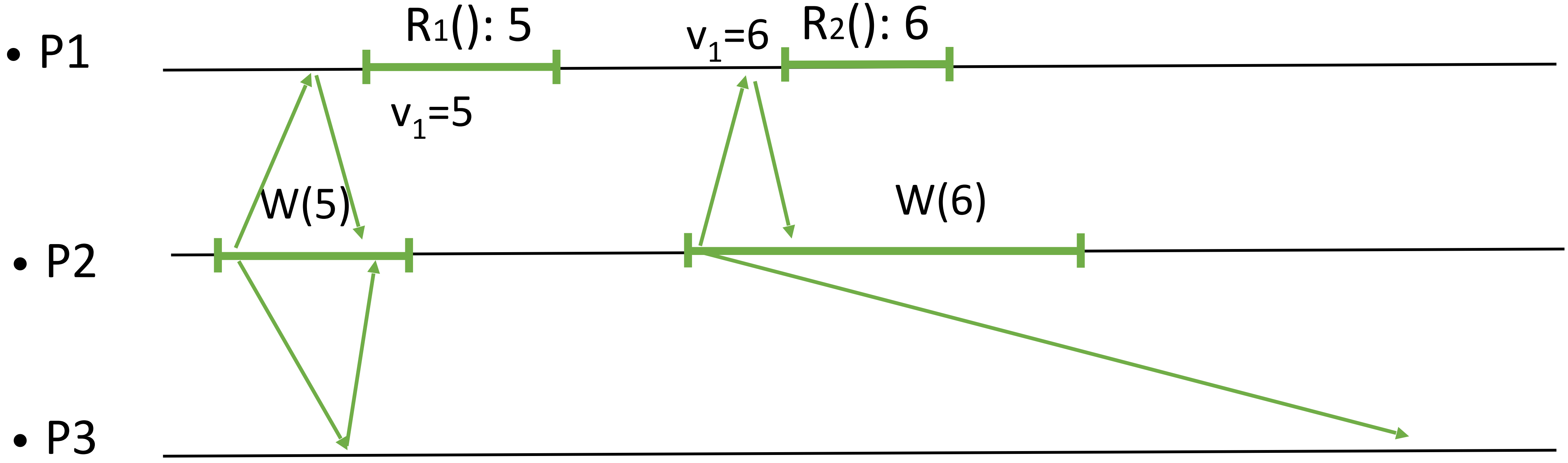
Atomicity?



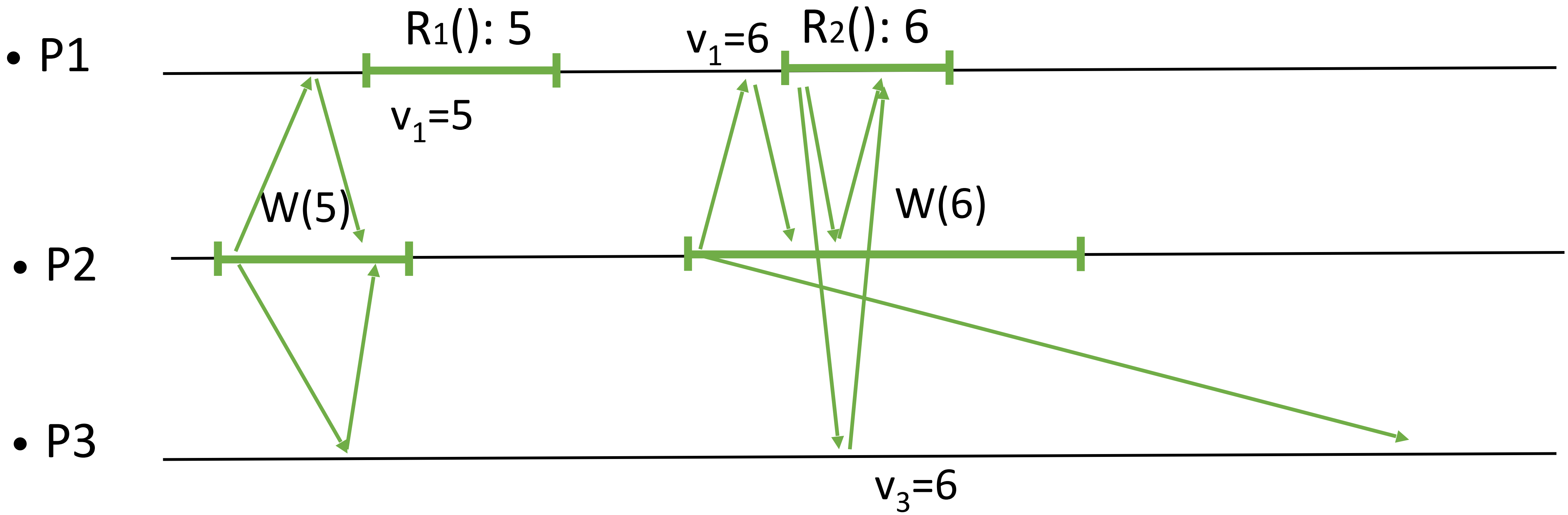
Atomicity?



Atomicity?

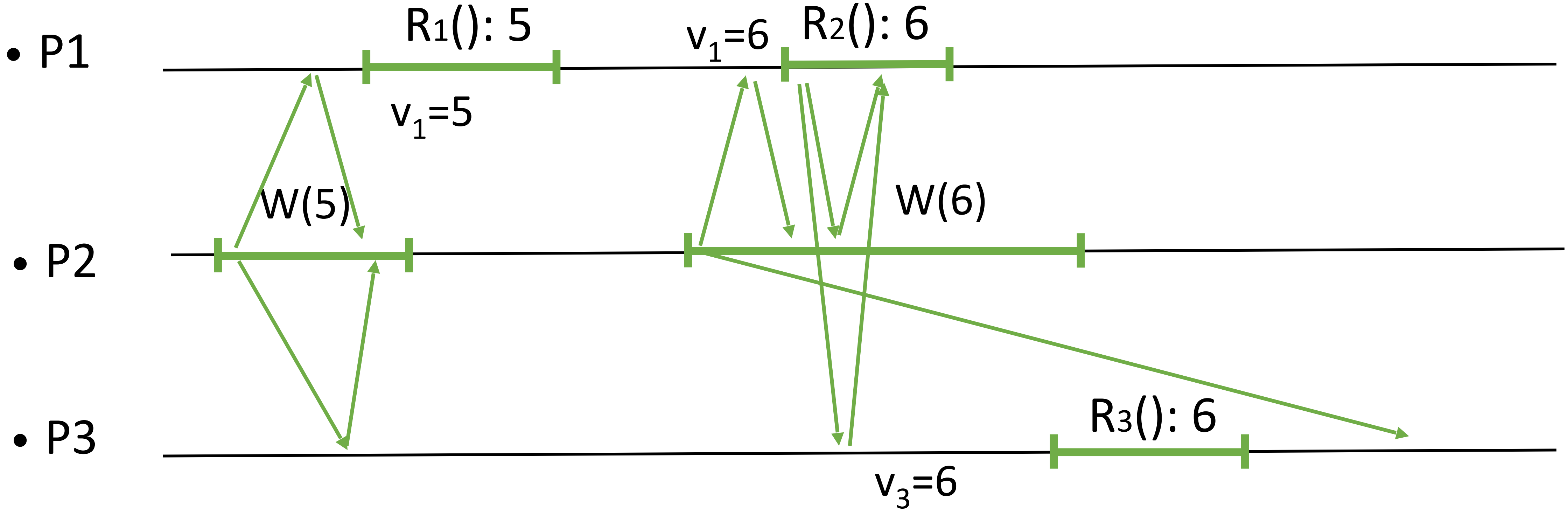


Atomicity?



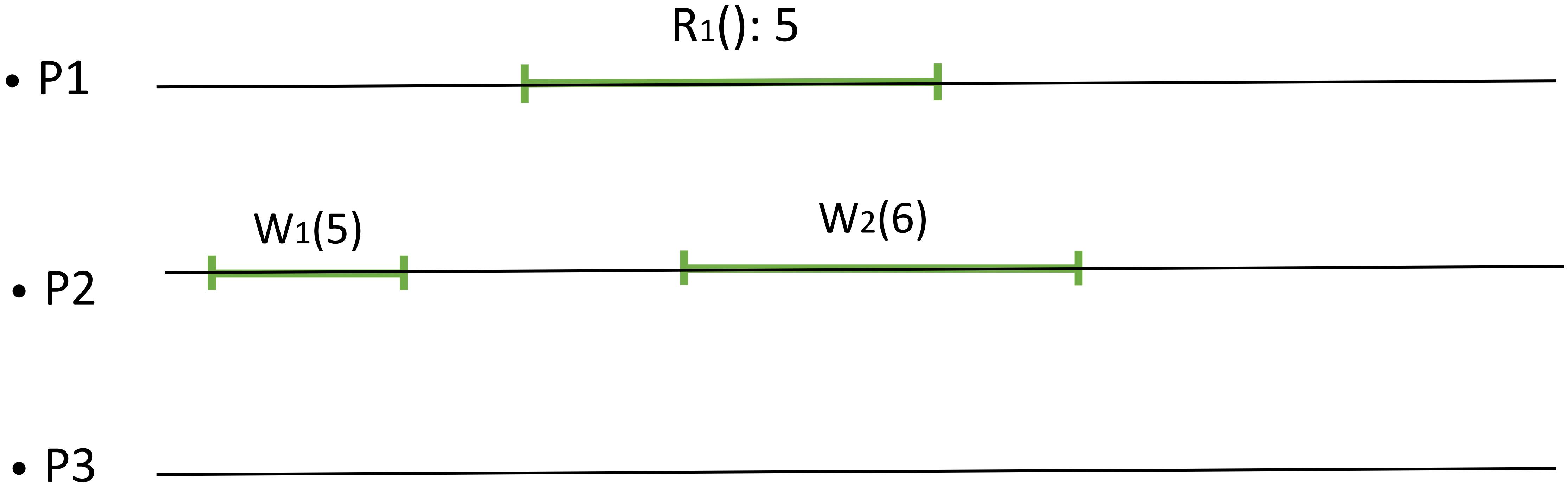
R2 that is returning the new value 6 makes sure that the other processes are updated.

Atomicity?



R2 that is returning the new value 6 makes sure that the other processes are updated.

Still a problem?



- P1

$R_1()$: 5

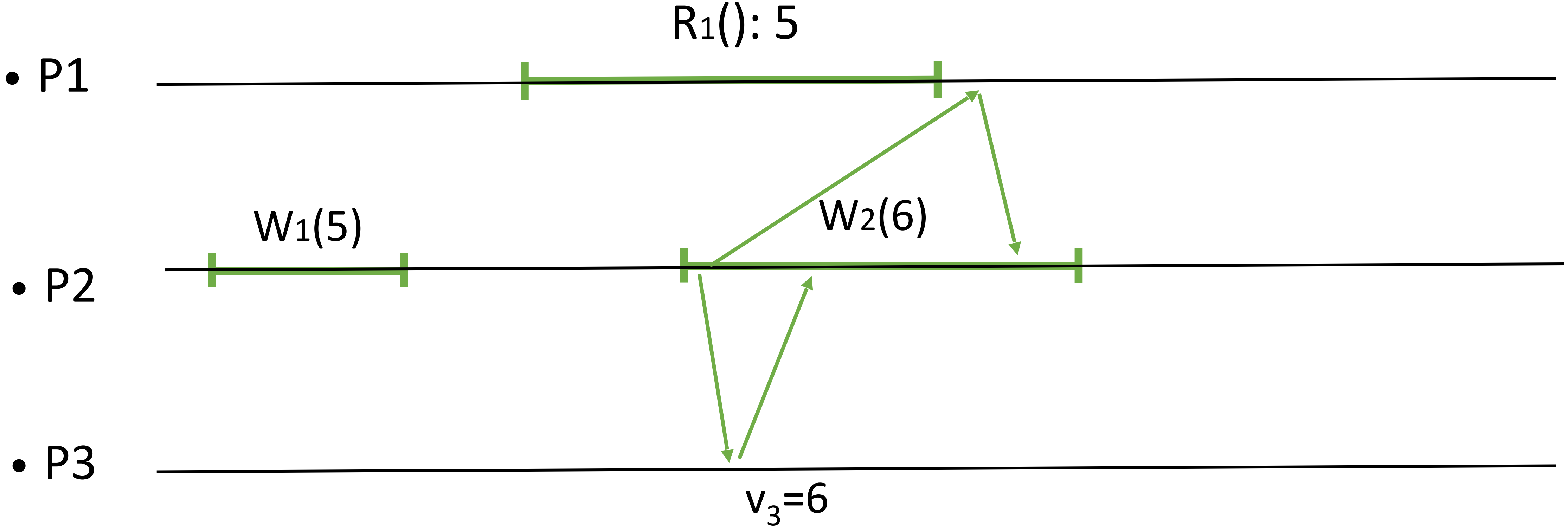
- P2

$W_1(5)$

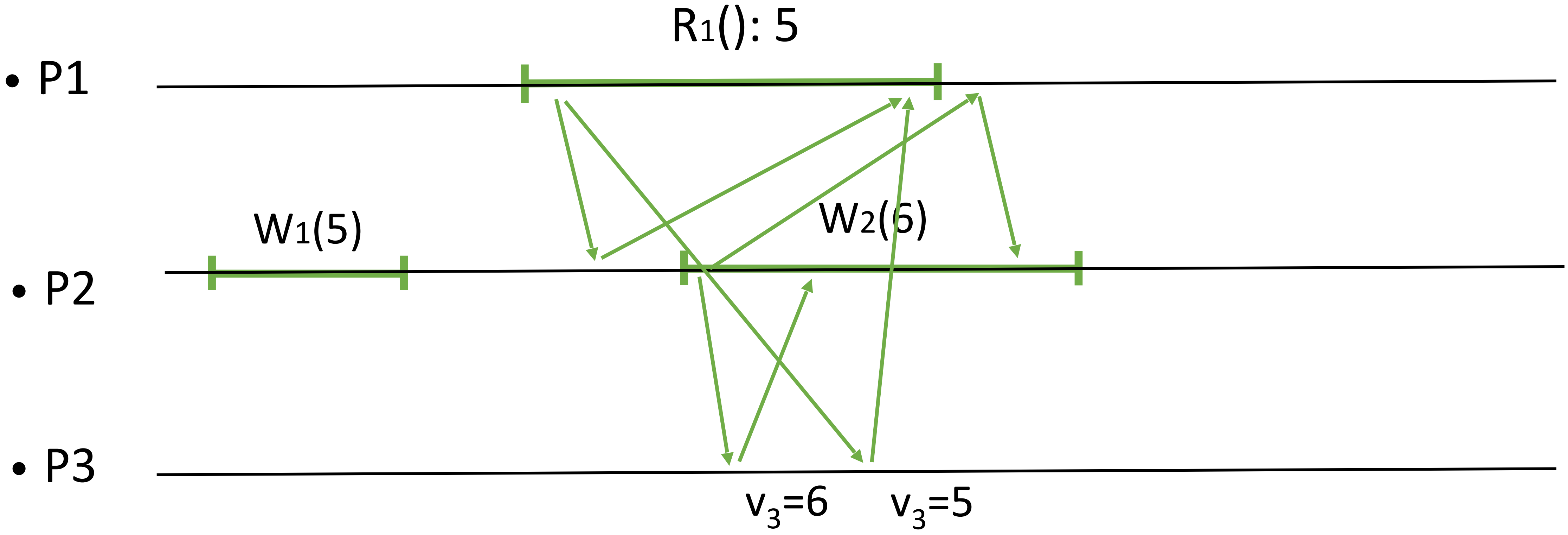
$W_2(6)$

- P3

Still a problem?

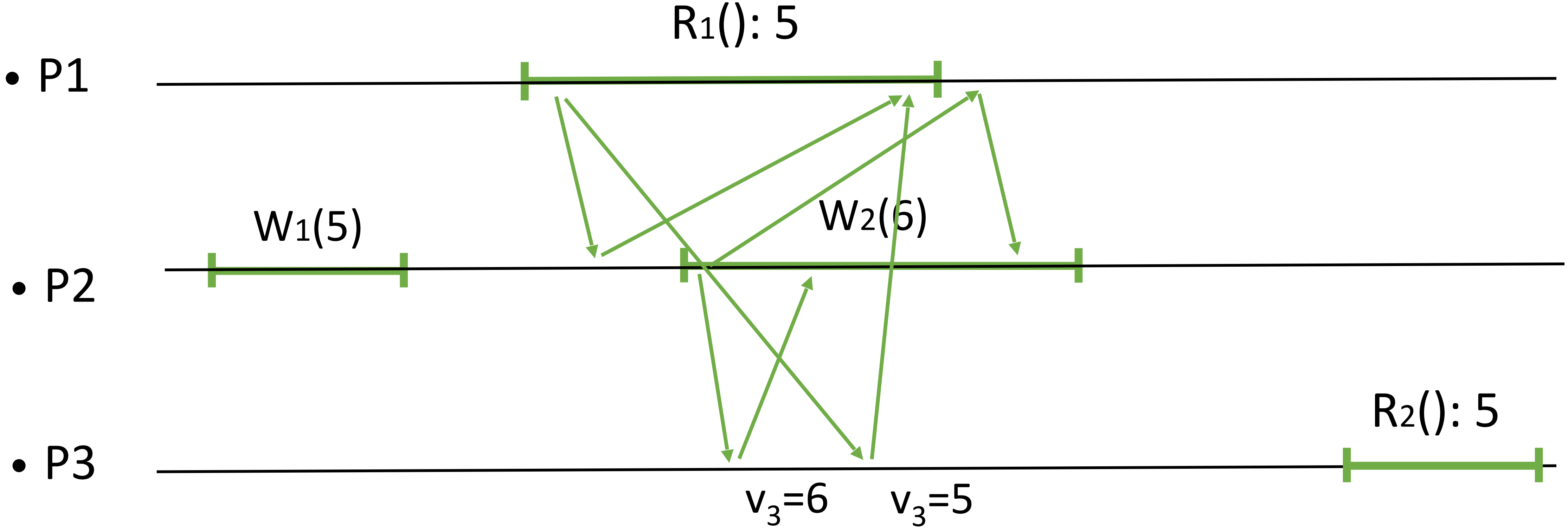


Still a problem?



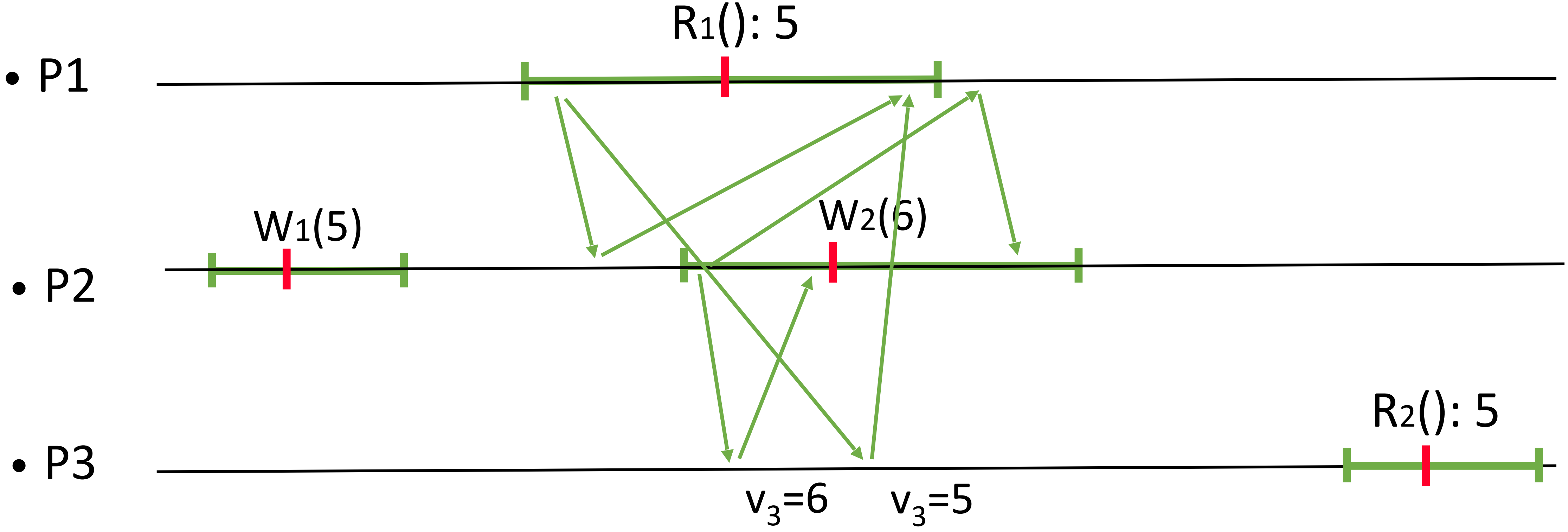
The updates by R1 overwrite the updates by W(6).
This is not linearizable. R2 should be linearized after W2.

Still a problem?



The updates by R1 overwrite the updates by W(6).
This is not linearizable. R2 should be linearized after W2.

Still a problem?



The updates by R1 overwrite the updates by W(6).
This is not linearizable. R2 should be linearized after W2.

R3 should return 6.

Overview of this lecture

1. A 1-1 atomic fail-stop algorithm
2. From regular to atomic
3. **A 1-N atomic fail-stop algorithm**
4. A N-N atomic fail-stop algorithm
5. From fail-stop to fail-silent

A fail-stop 1-N algorithm

Idea:

- Write only newer values.
- The writer, p_1 maintains and propagates a timestamp ts_1
- Every process maintains a sequence number in addition to the local value of the register.

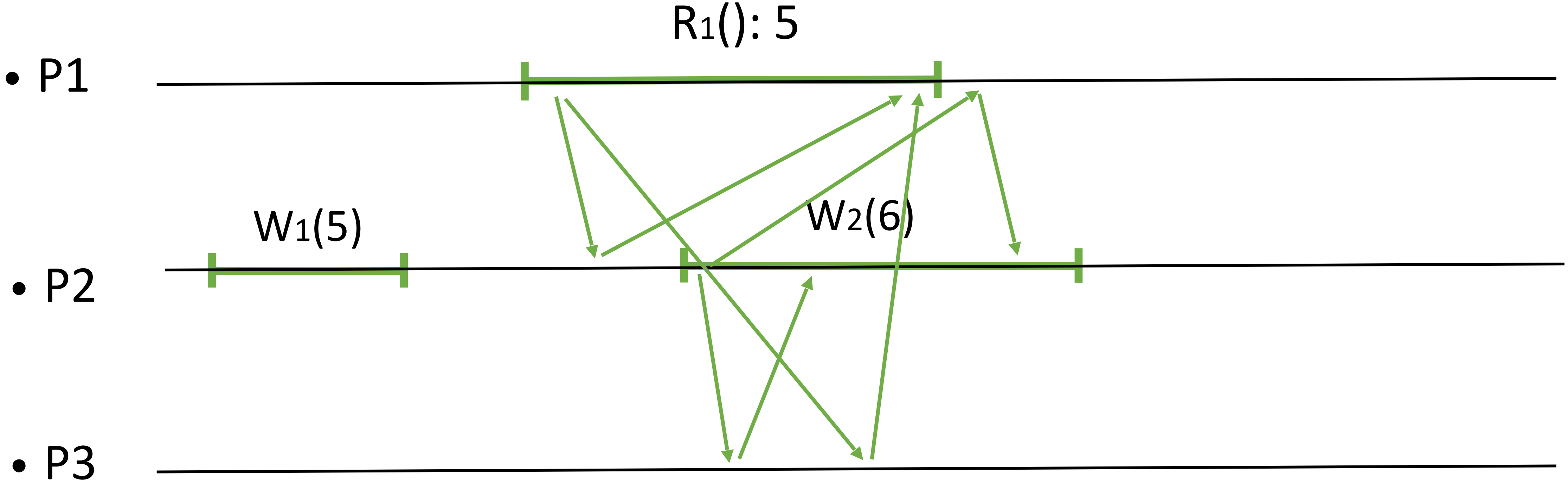
A fail-stop 1-N algorithm

```
**upon** Write(v) at $p_1$
  $ts_1 = ts_1 + 1$
  **trigger** send [W,$ts_1$,v] to all
  **foreach** $p_i$, wait until either:
     deliver [ack] or
     suspect [$p_i$]
  **trigger** ok
```

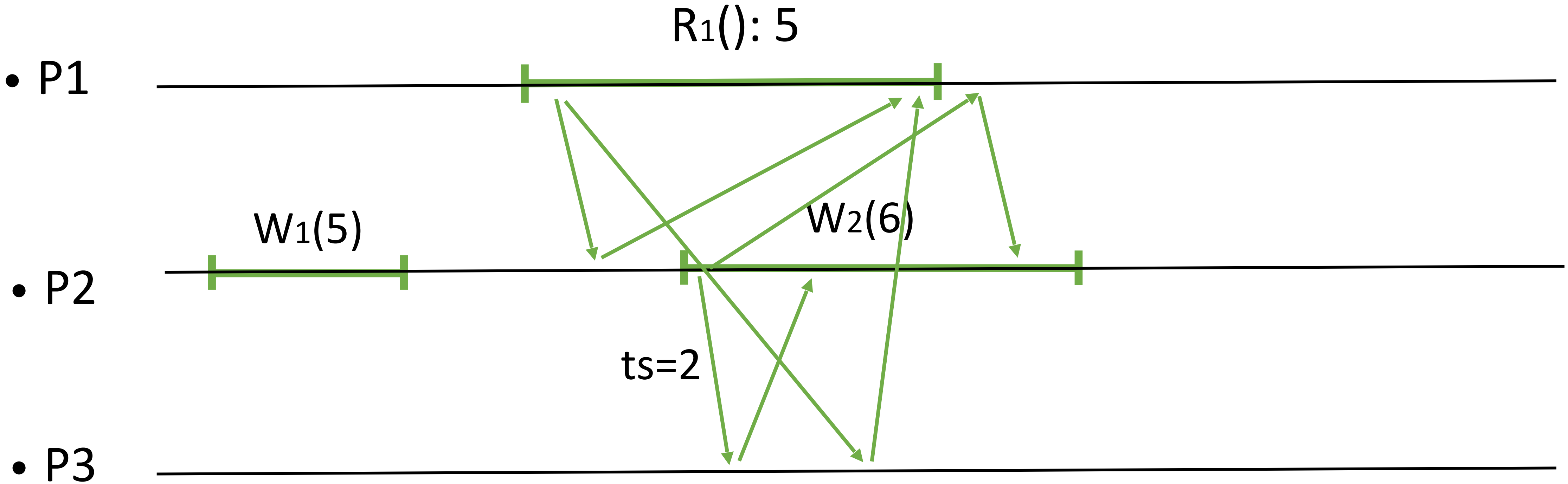
```
**upon** deliver [W,ts,v] from $p_j$
   **if** ts > $sn_i$ **then**
        $v_i$ := v
        $sn_i$ := ts
   **trigger** send [ack] to $p_j$
```

```
**upon** Read() at $p_i$
   **trigger** send [W,$sn_i$,$v_i$] to all
   **foreach** $p_j$, wait until either:
      deliver [ack] or
      suspect [$p_j$]
   **trigger** Ret($v_i$)
```

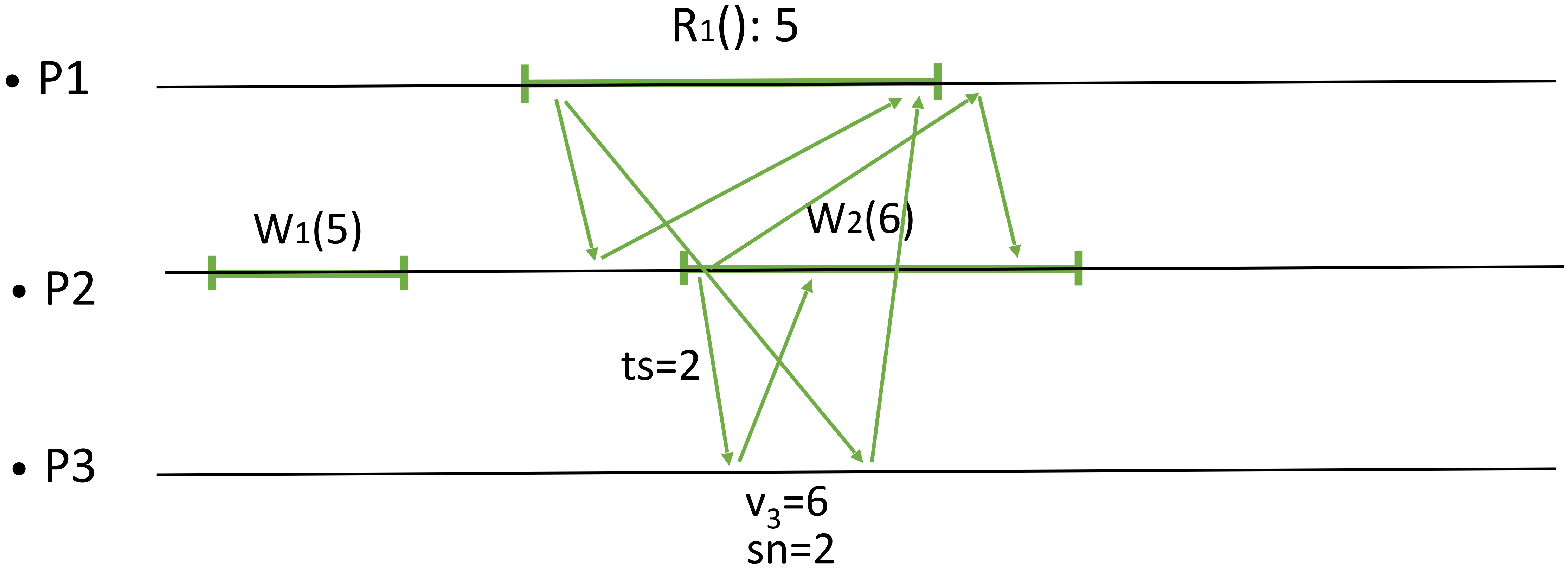
Still a problem?



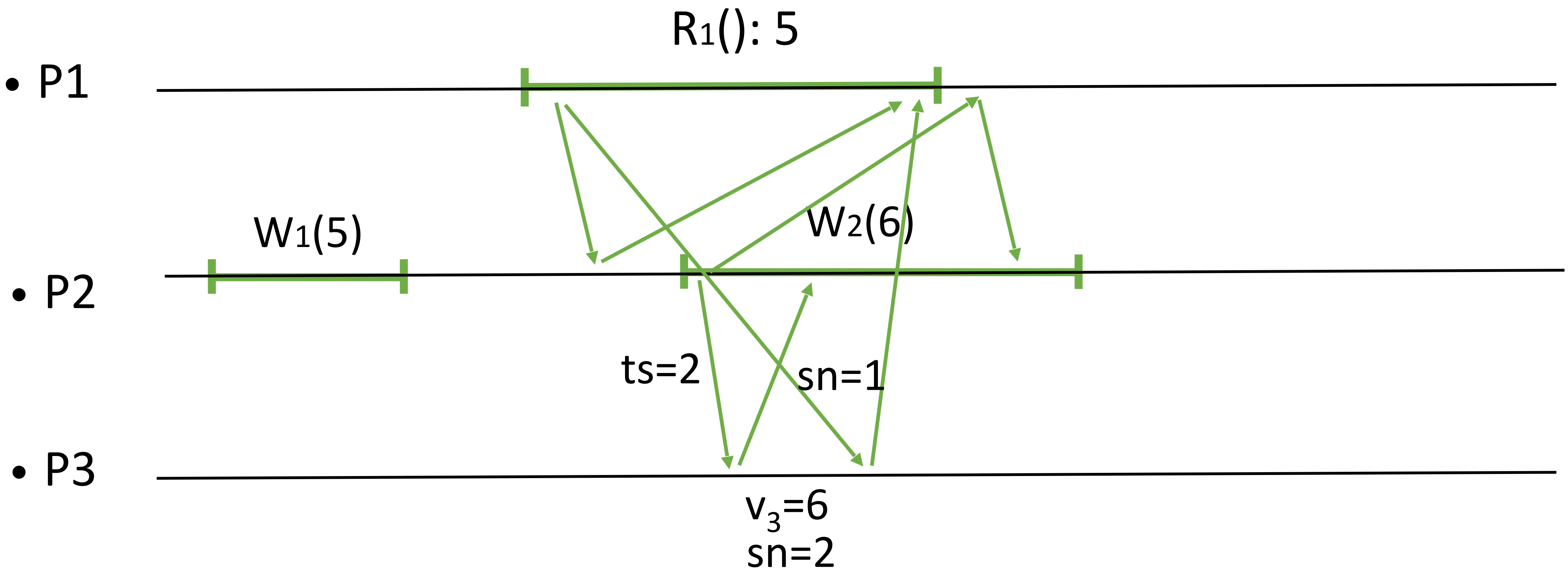
Still a problem?



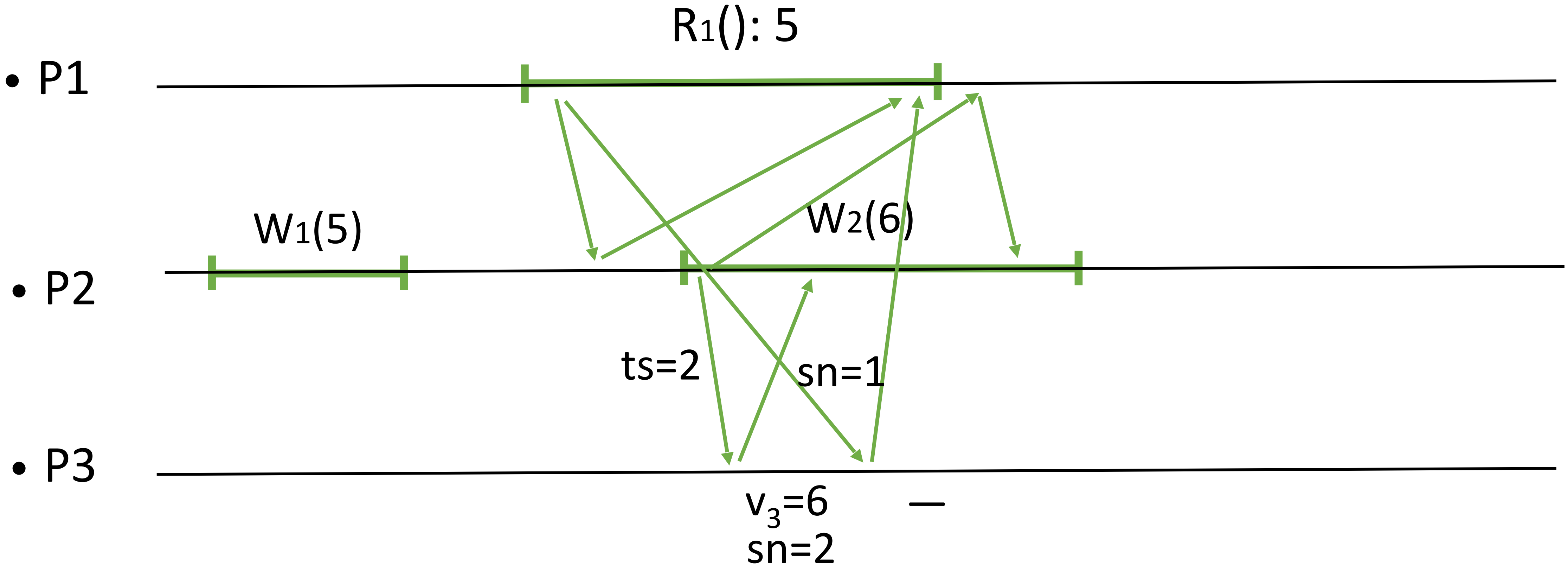
Still a problem?



Still a problem?

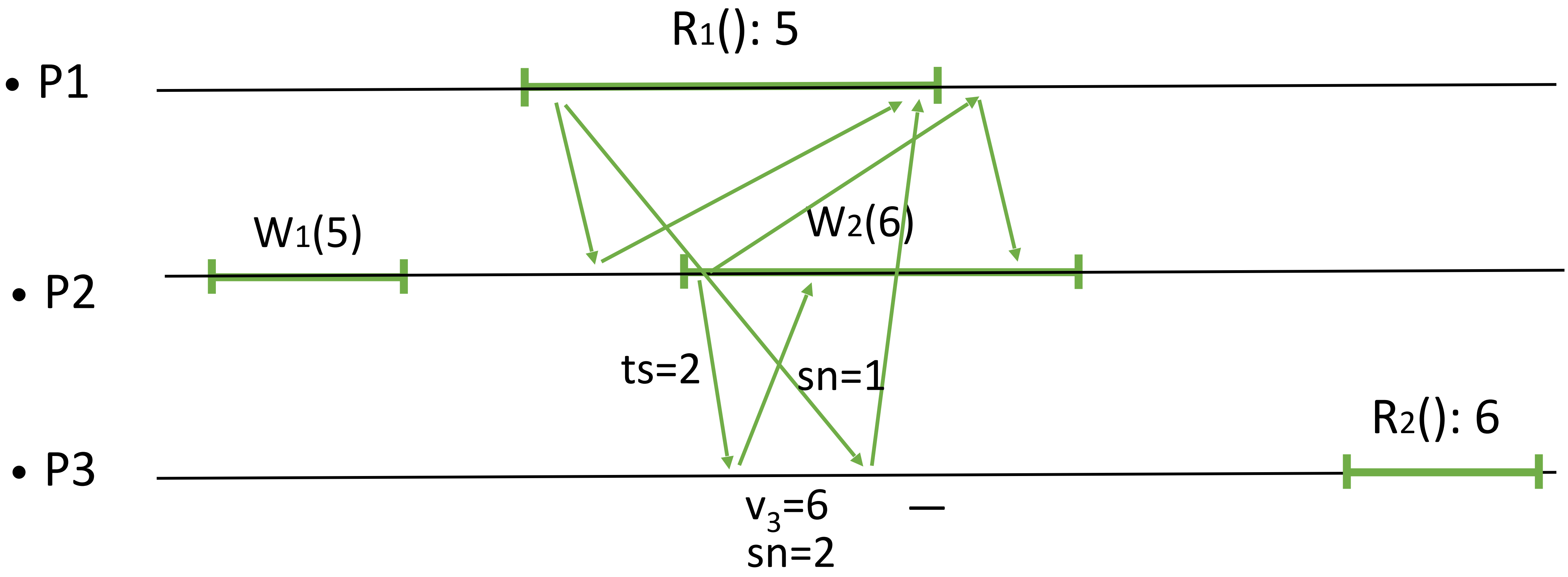


Still a problem?



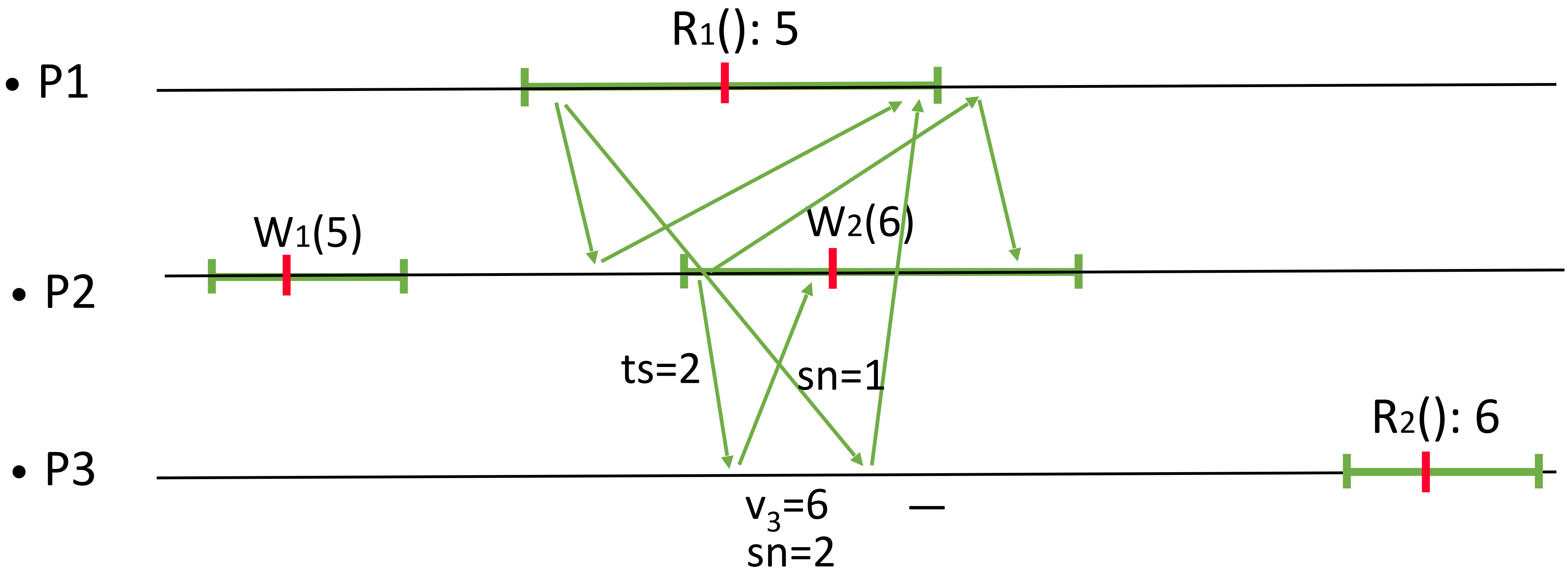
The updates by R1 cannot overwrite the updates by W2.

Still a problem?



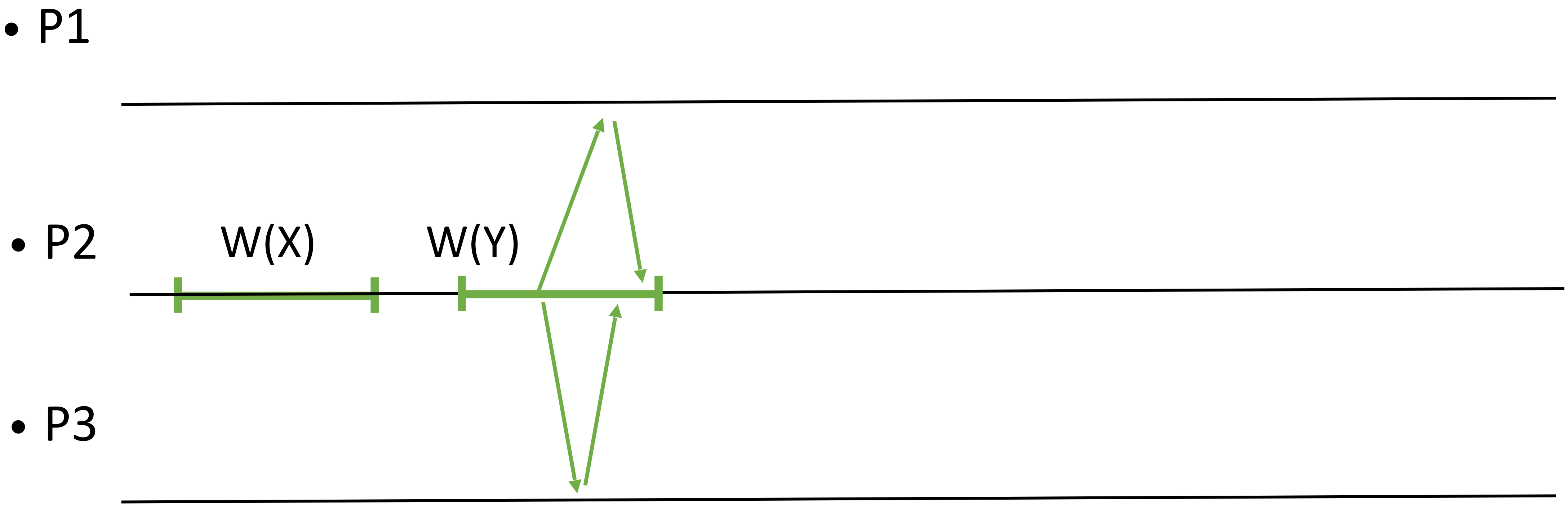
The updates by R1 cannot overwrite the updates by W2.

Still a problem?



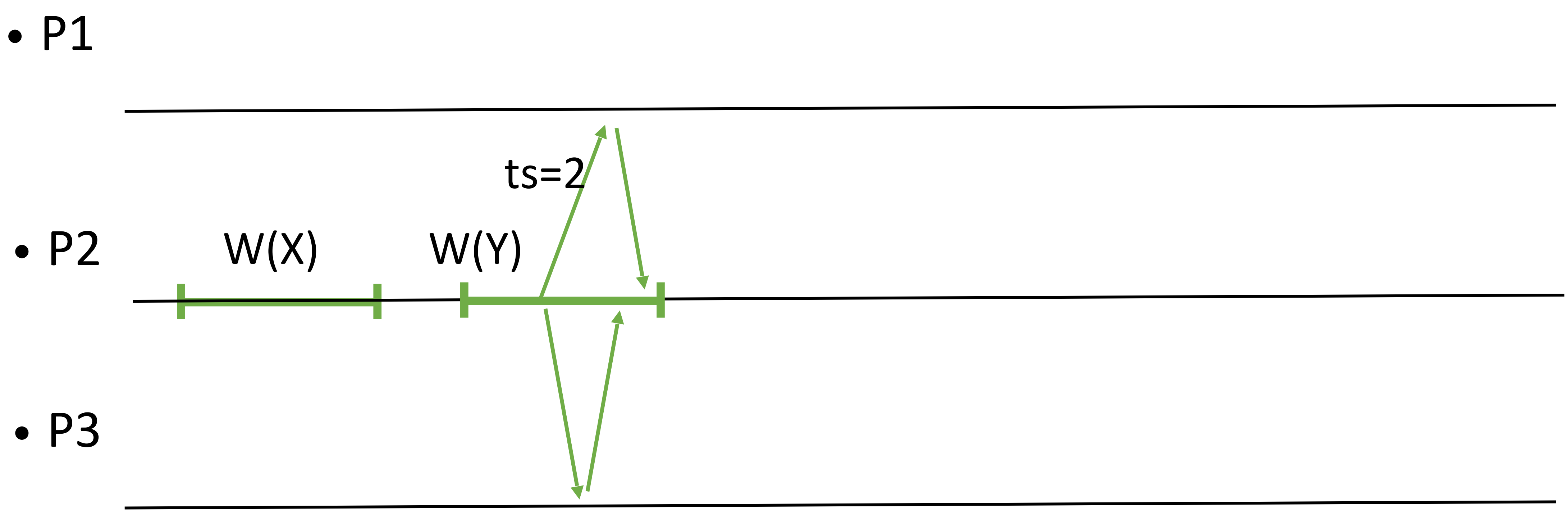
The updates by R1 cannot overwrite the updates by W2.

Why not N-N?



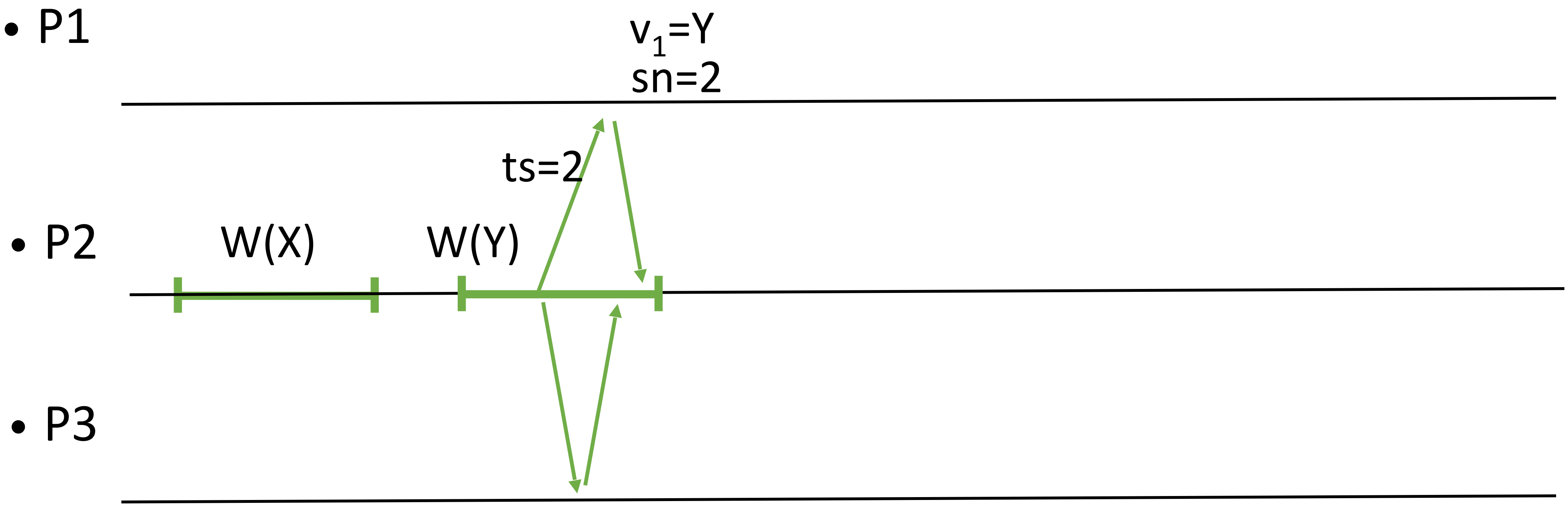
The updates from W(Z) have timestamp 1. The updates from W(Y) have timestamp 2. In P1, Z cannot overwrite Y.

Why not N-N?



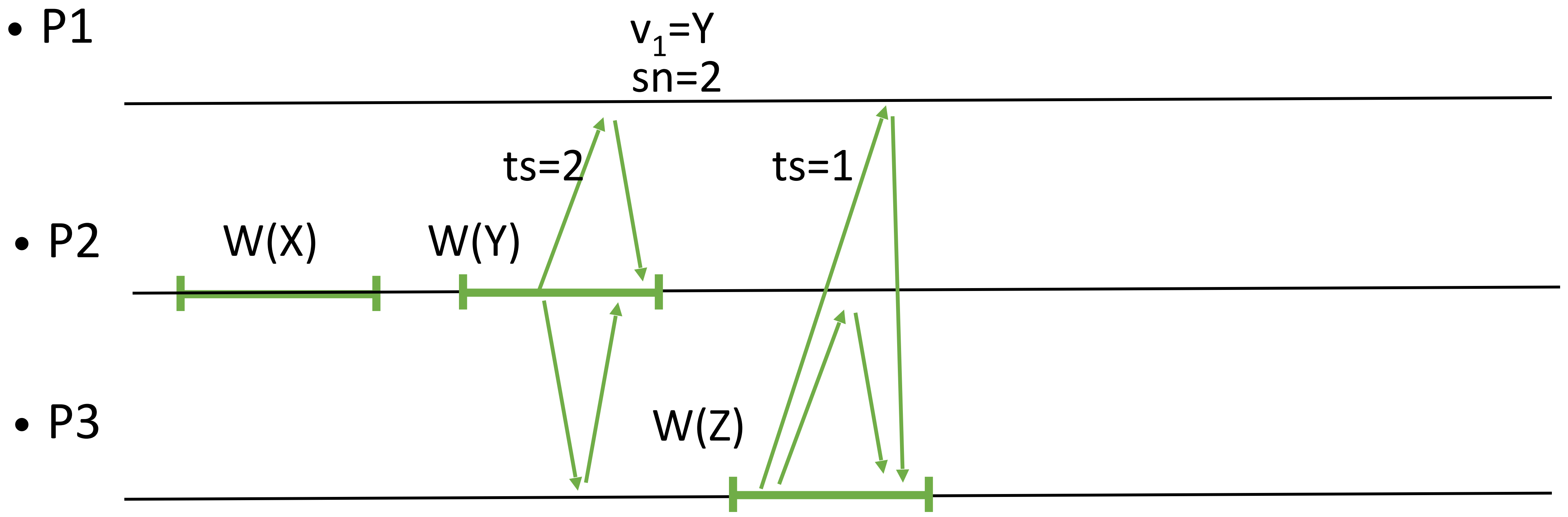
The updates from W(Z) have timestamp 1. The updates from W(Y) have timestamp 2. In P1, Z cannot overwrite Y.

Why not N-N?



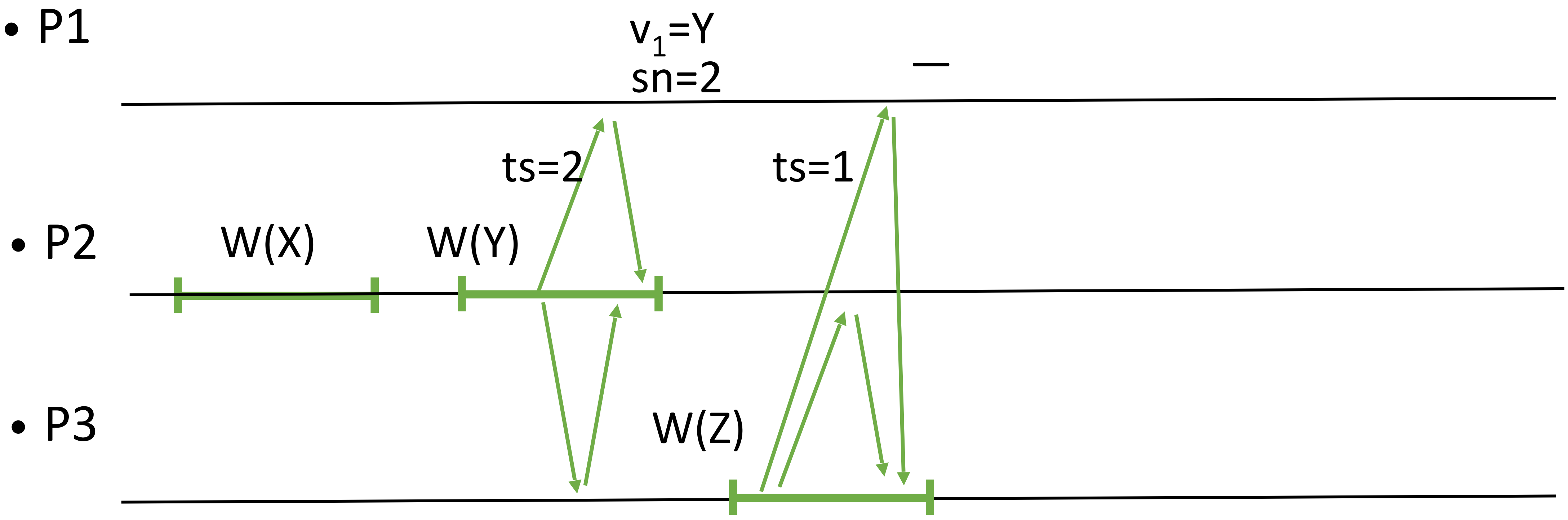
The updates from W(Z) have timestamp 1. The updates from W(Y) have timestamp 2. In P1, Z cannot overwrite Y.

Why not N-N?



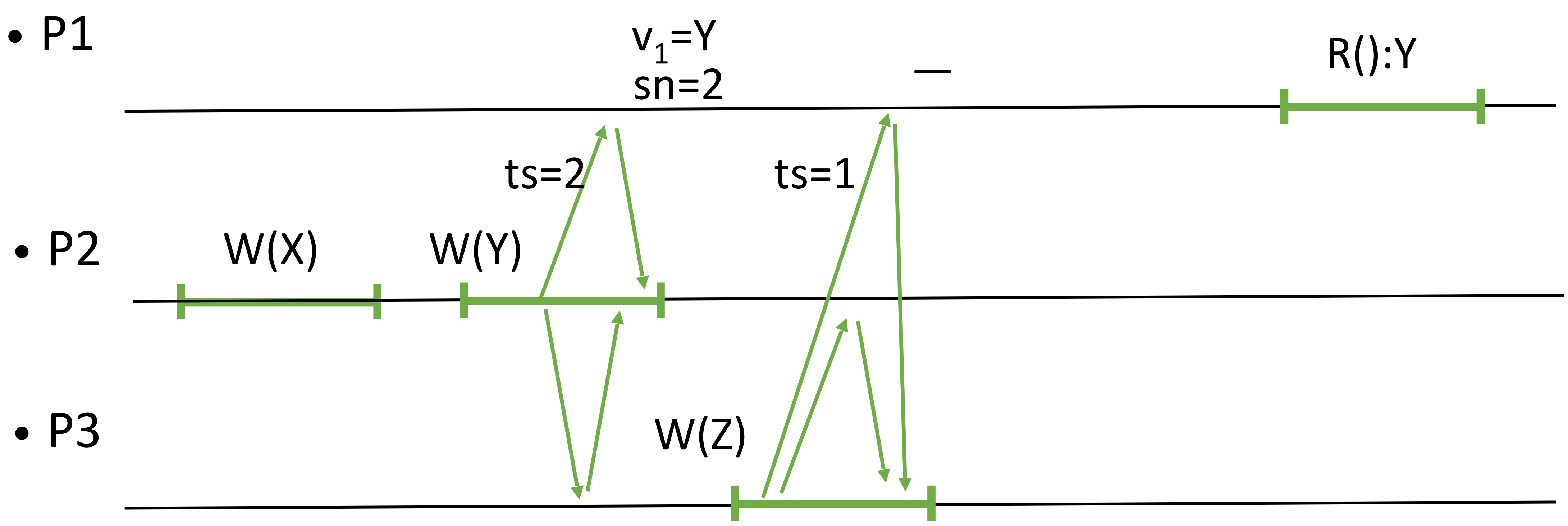
The updates from W(Z) have timestamp 1. The updates from W(Y) have timestamp 2. In P1, Z cannot overwrite Y.

Why not N-N?



The updates from W(Z) have timestamp 1. The updates from W(Y) have timestamp 2. In P1, Z cannot overwrite Y.

Why not N-N?



The updates from W(Z) have timestamp 1. The updates from W(Y) have timestamp 2. In P1, Z cannot overwrite Y.

Overview of this lecture

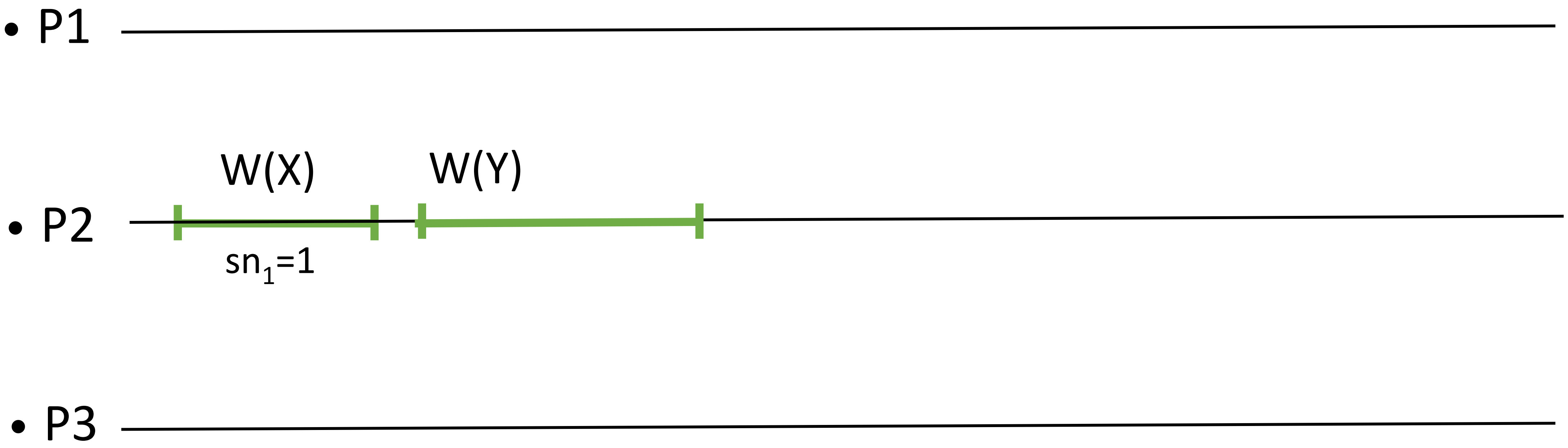
1. A 1-1 atomic fail-stop algorithm
2. From regular to atomic
3. A 1-N atomic fail-stop algorithm
4. **A N-N atomic fail-stop algorithm**
5. From fail-stop to fail-silent

A fail-stop N-N algorithm

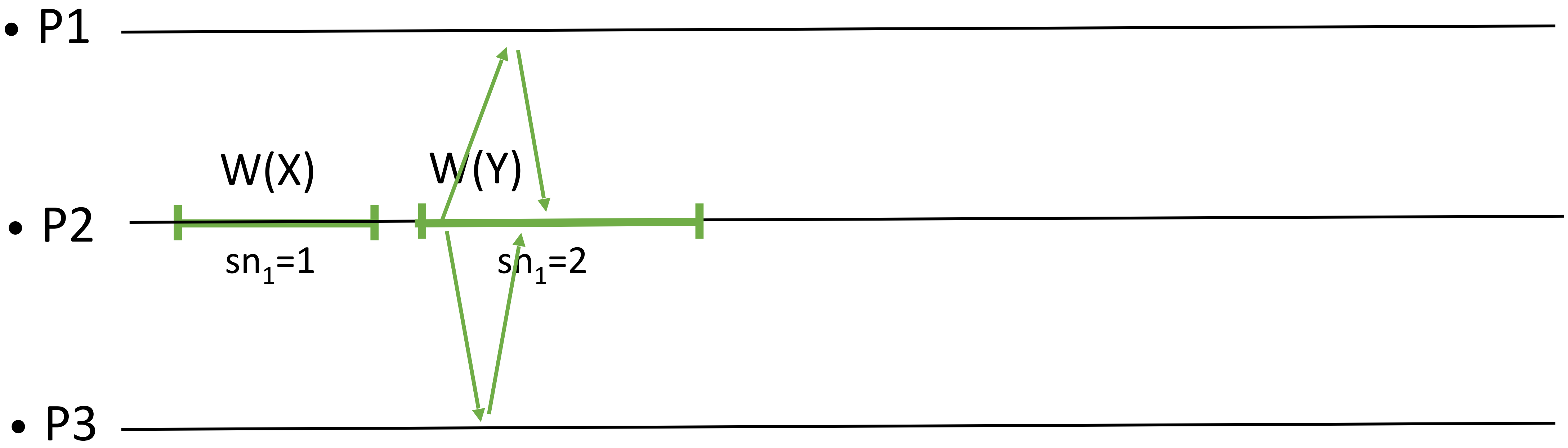
Idea:

- To write, first collect the largest timestamp, and increment it.

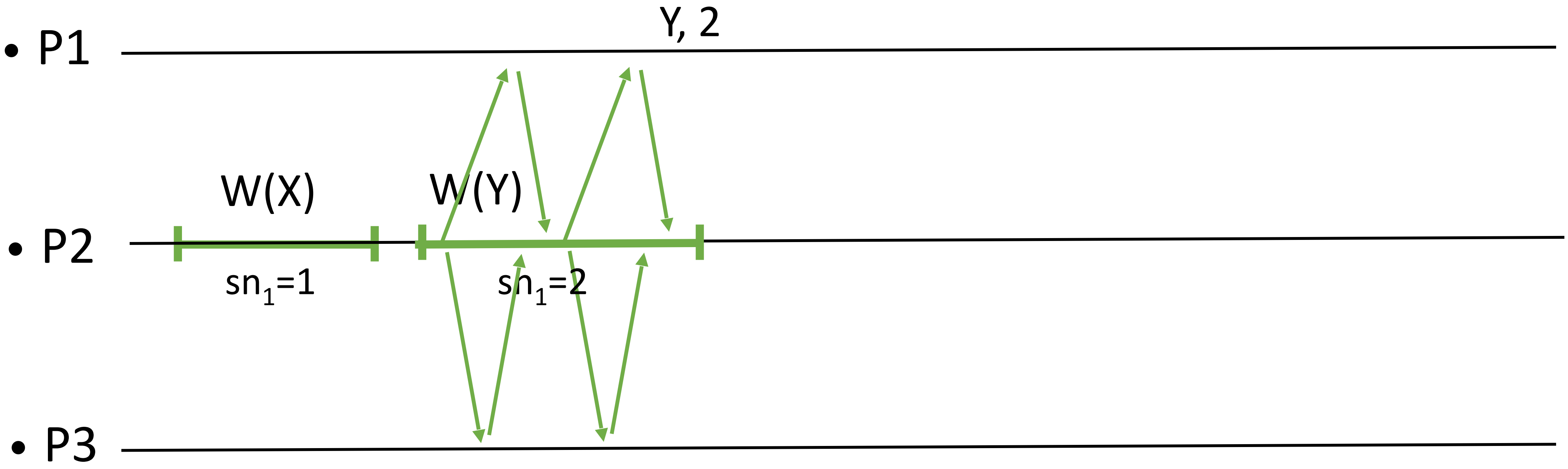
Now N-N?



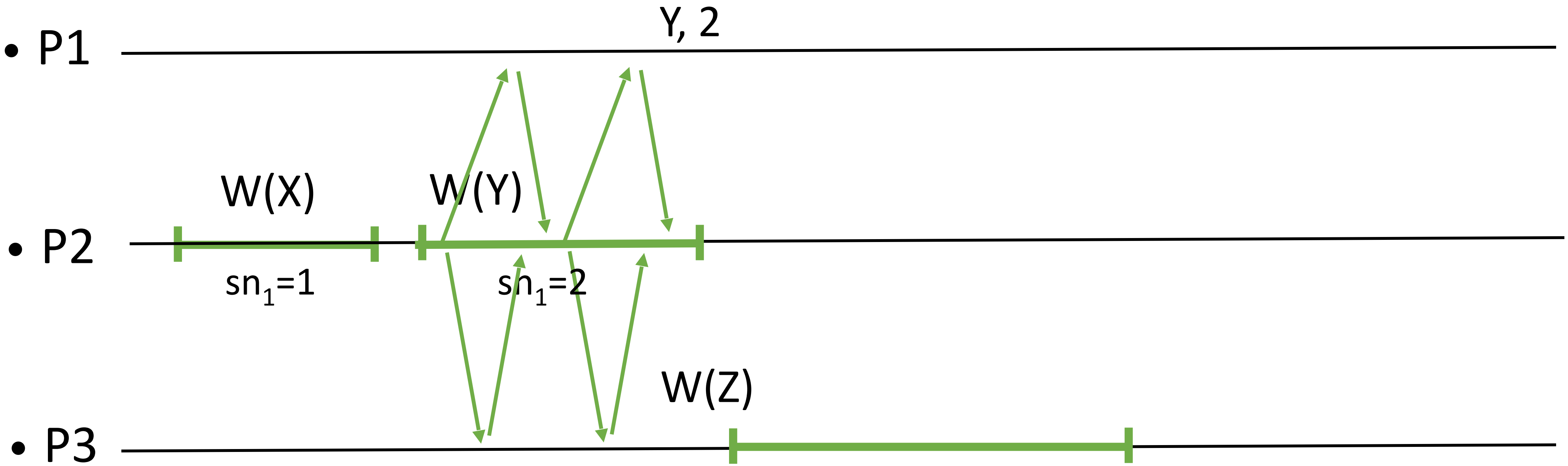
Now N-N?



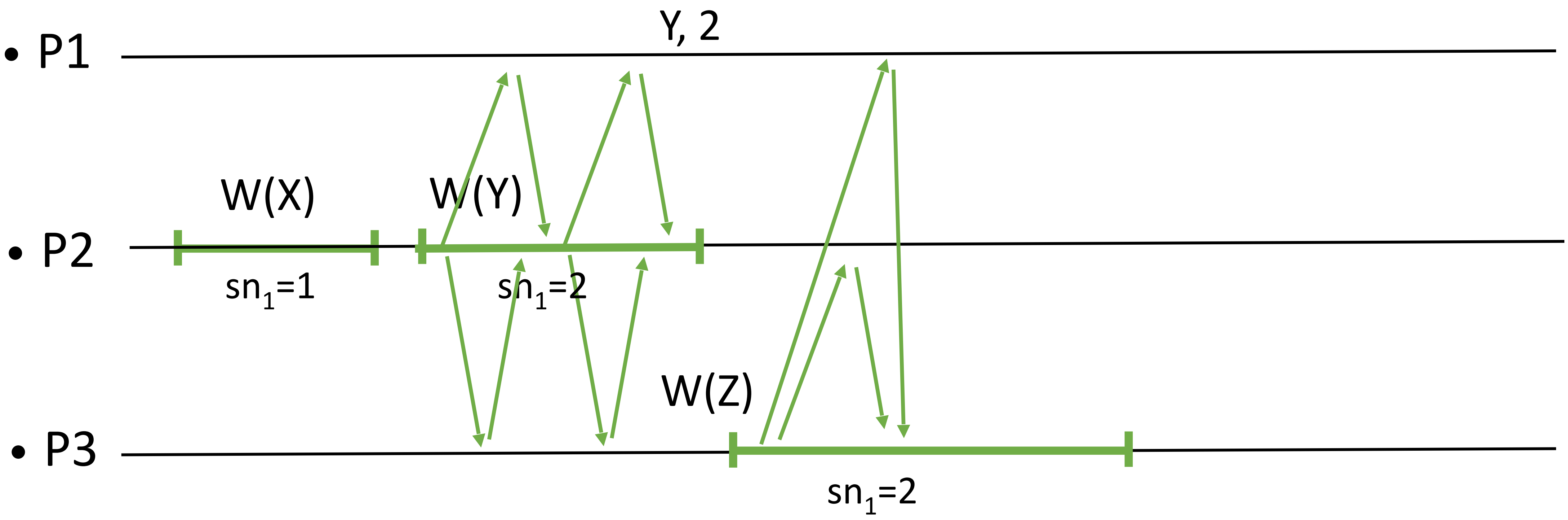
Now N-N?



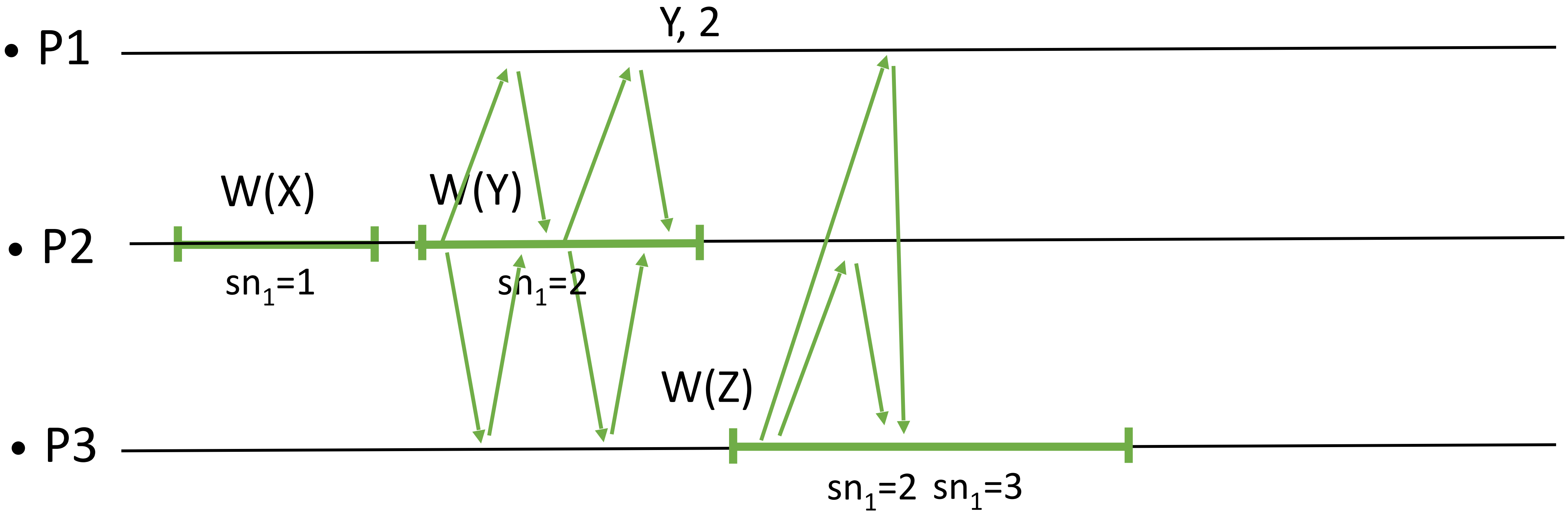
Now N-N?



Now N-N?

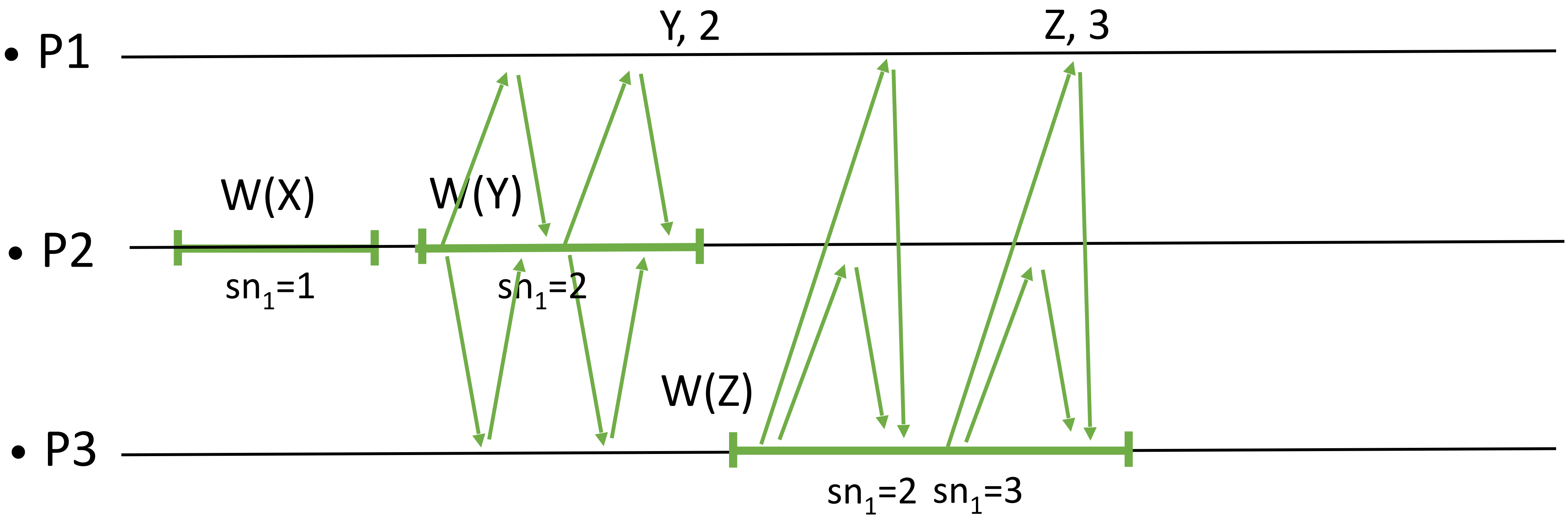


Now N-N?



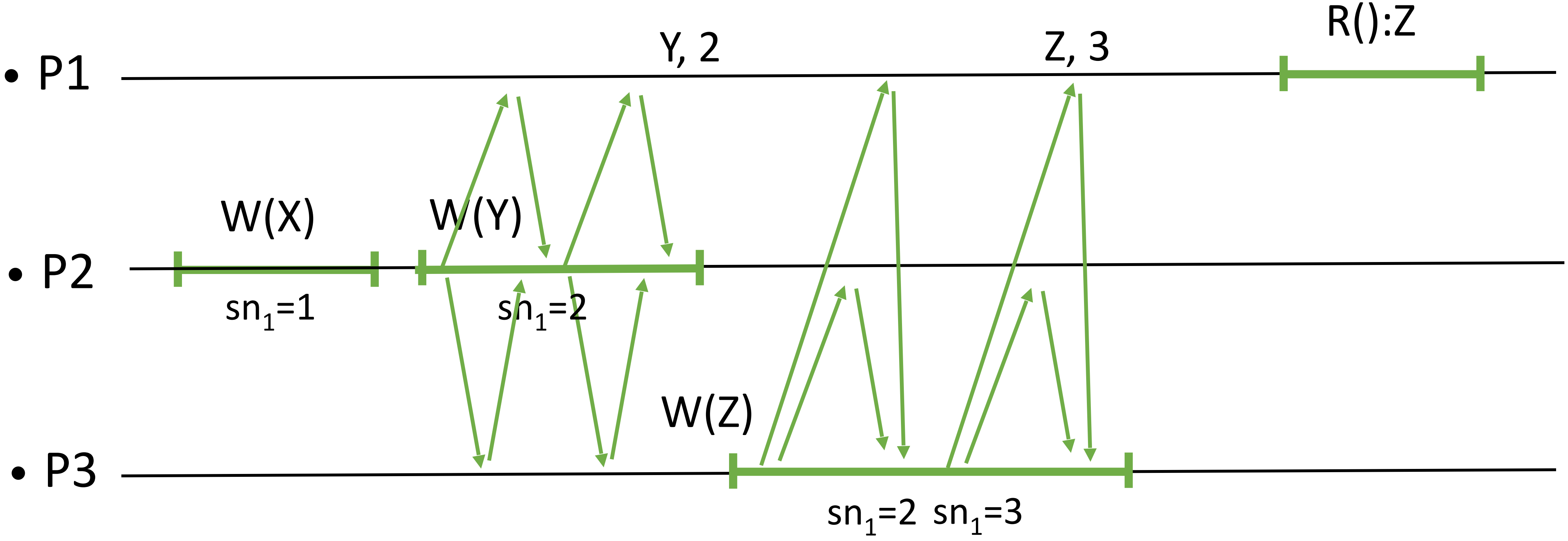
W(Z) collects the largest timestamp 2 and sends updates with timestamp 3.

Now N-N?



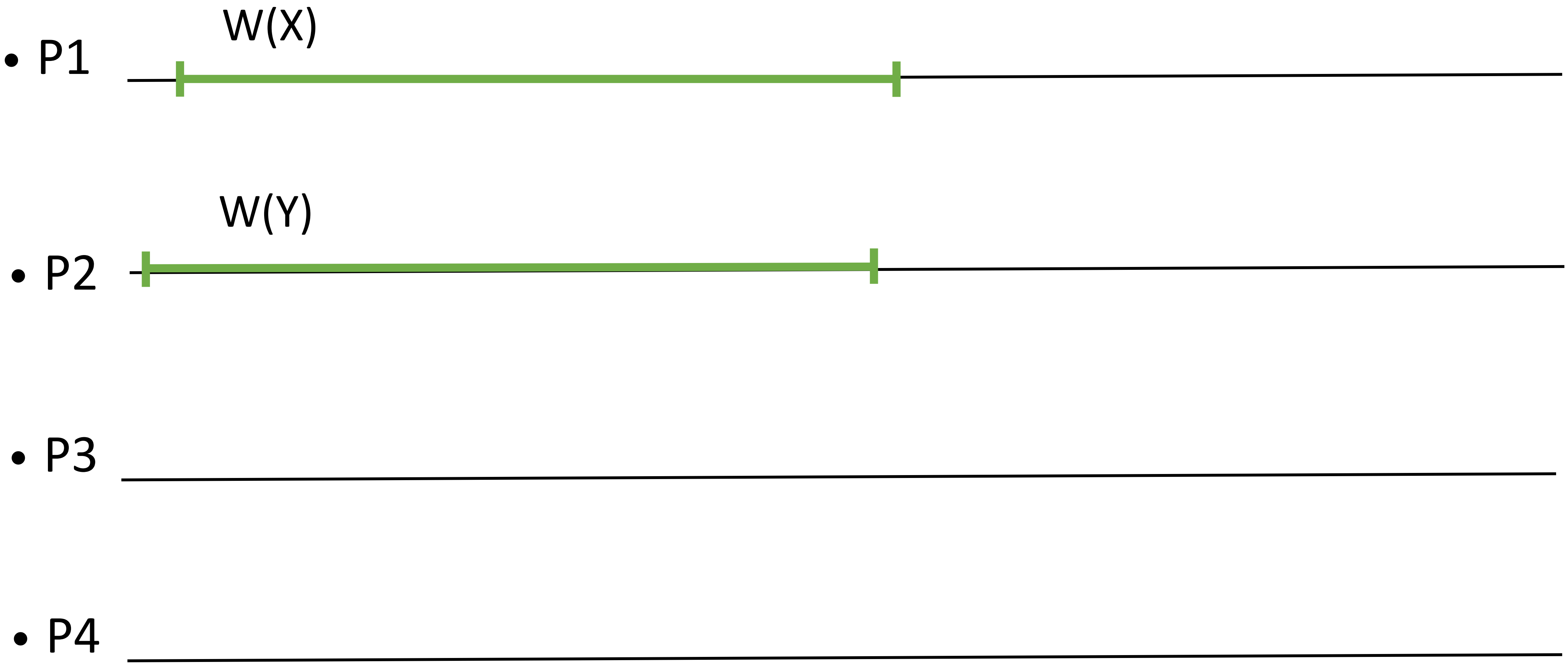
W(Z) collects the largest timestamp 2 and sends updates with timestamp 3.

Now N-N?



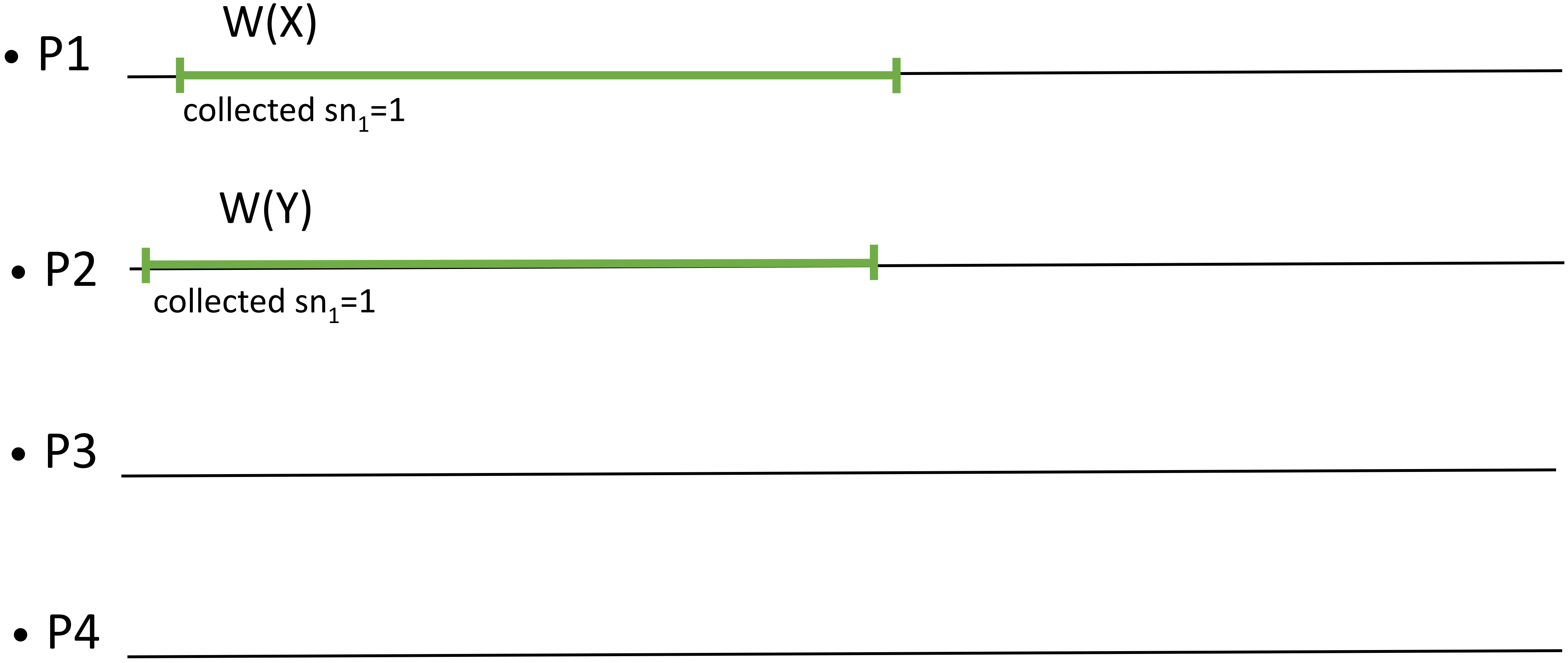
W(Z) collects the largest timestamp 2 and sends updates with timestamp 3.

Timestamp not enough?

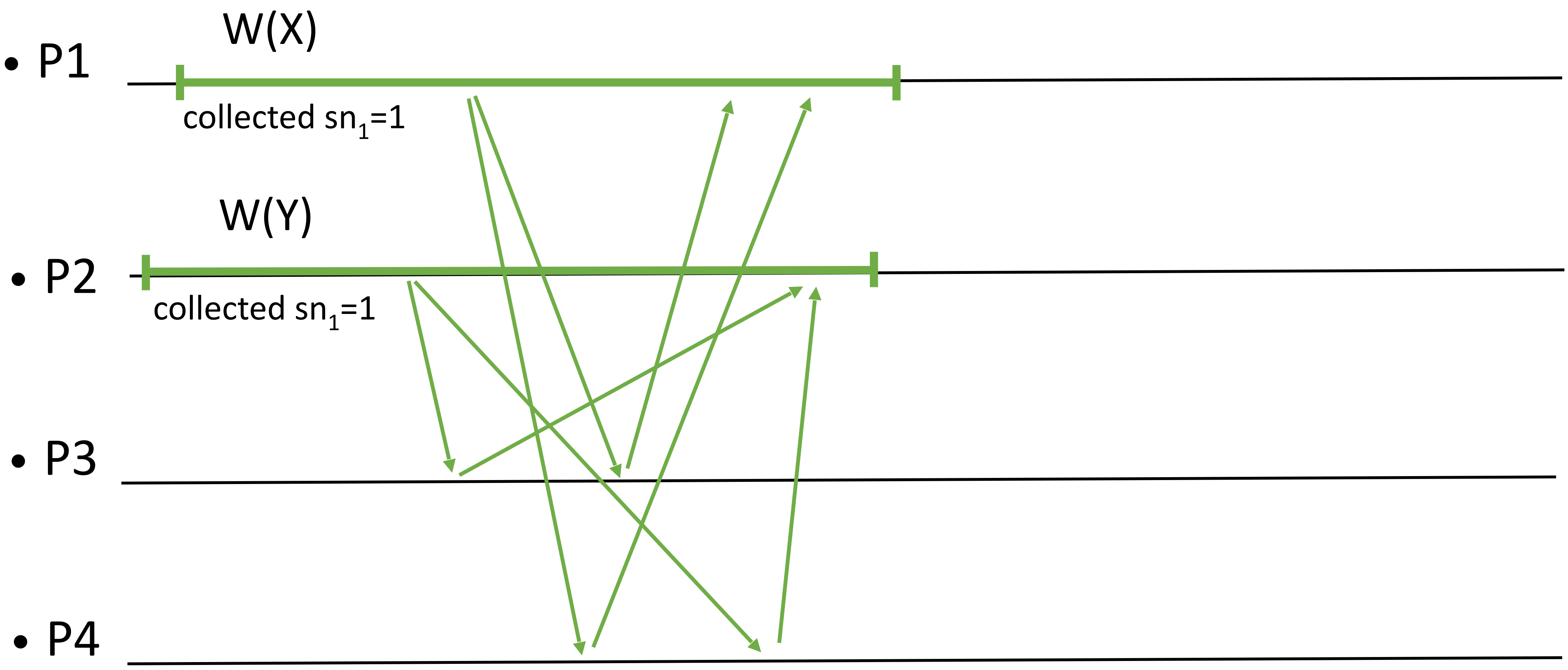


- P1
- P2
- P3
- P4

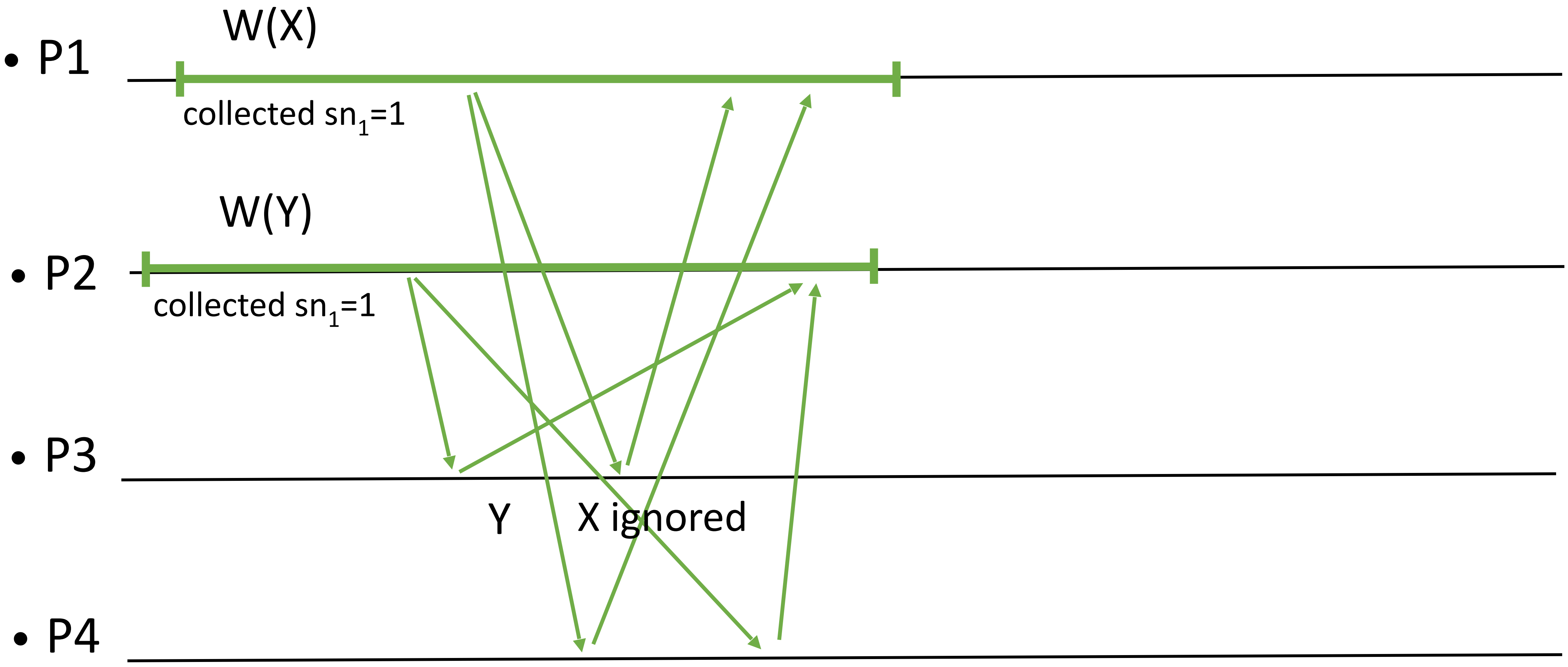
Timestamp not enough?



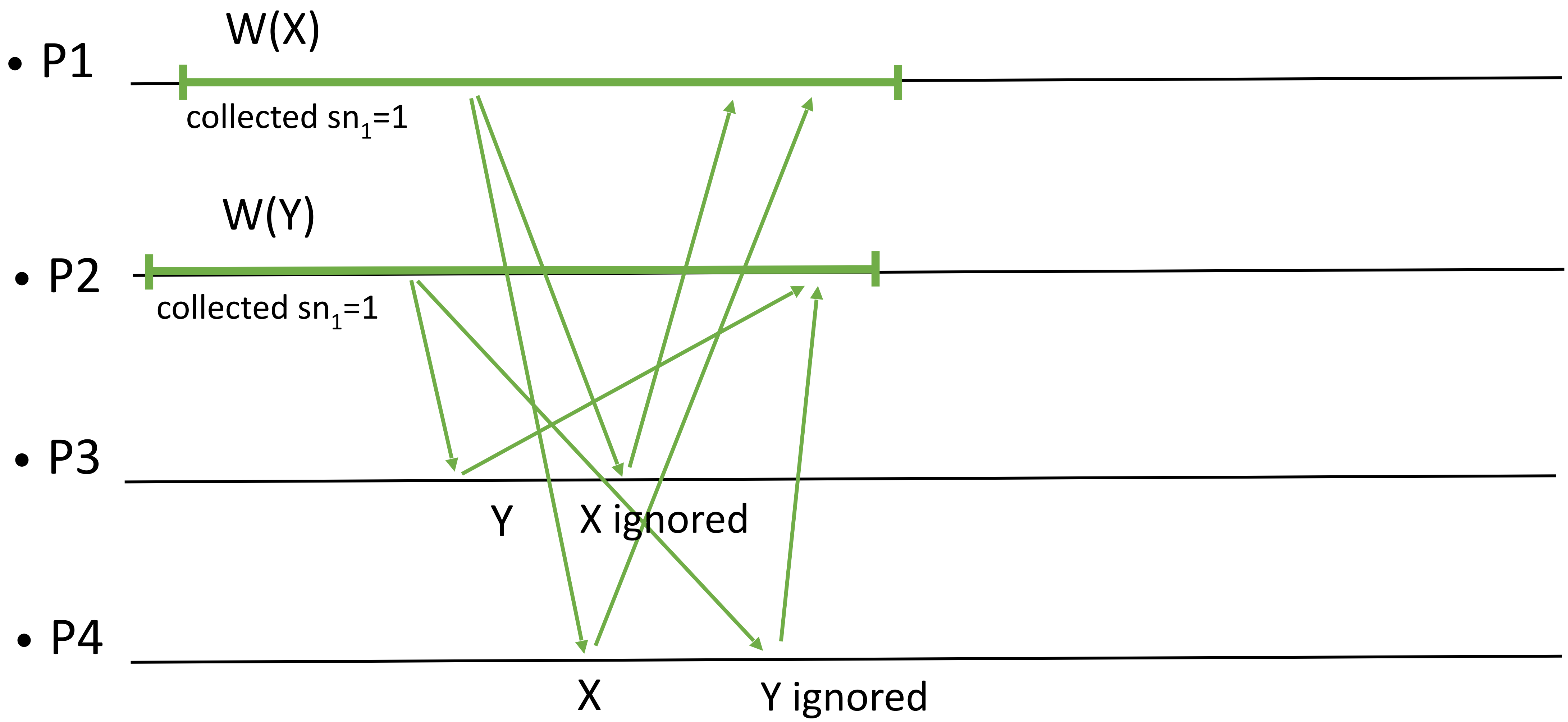
Timestamp not enough?



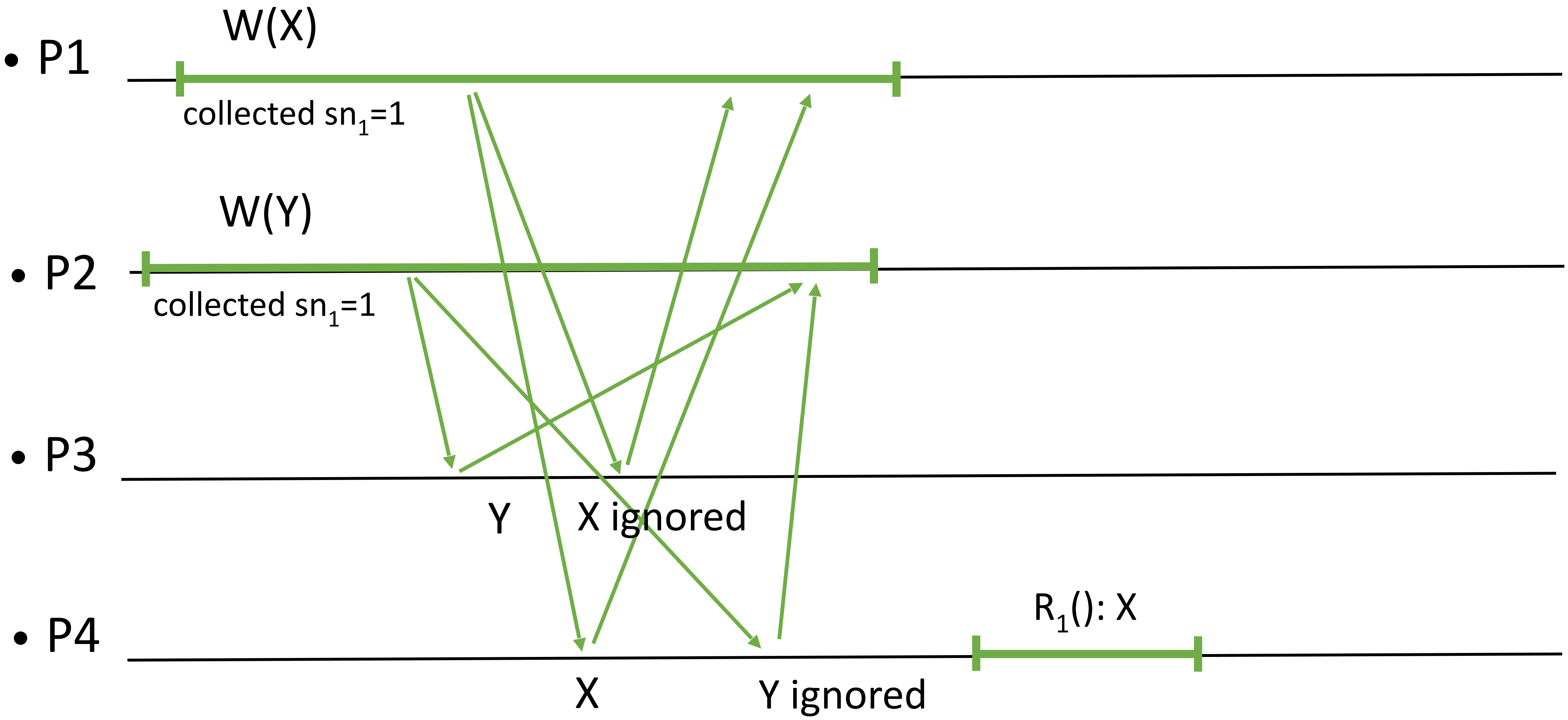
Timestamp not enough?



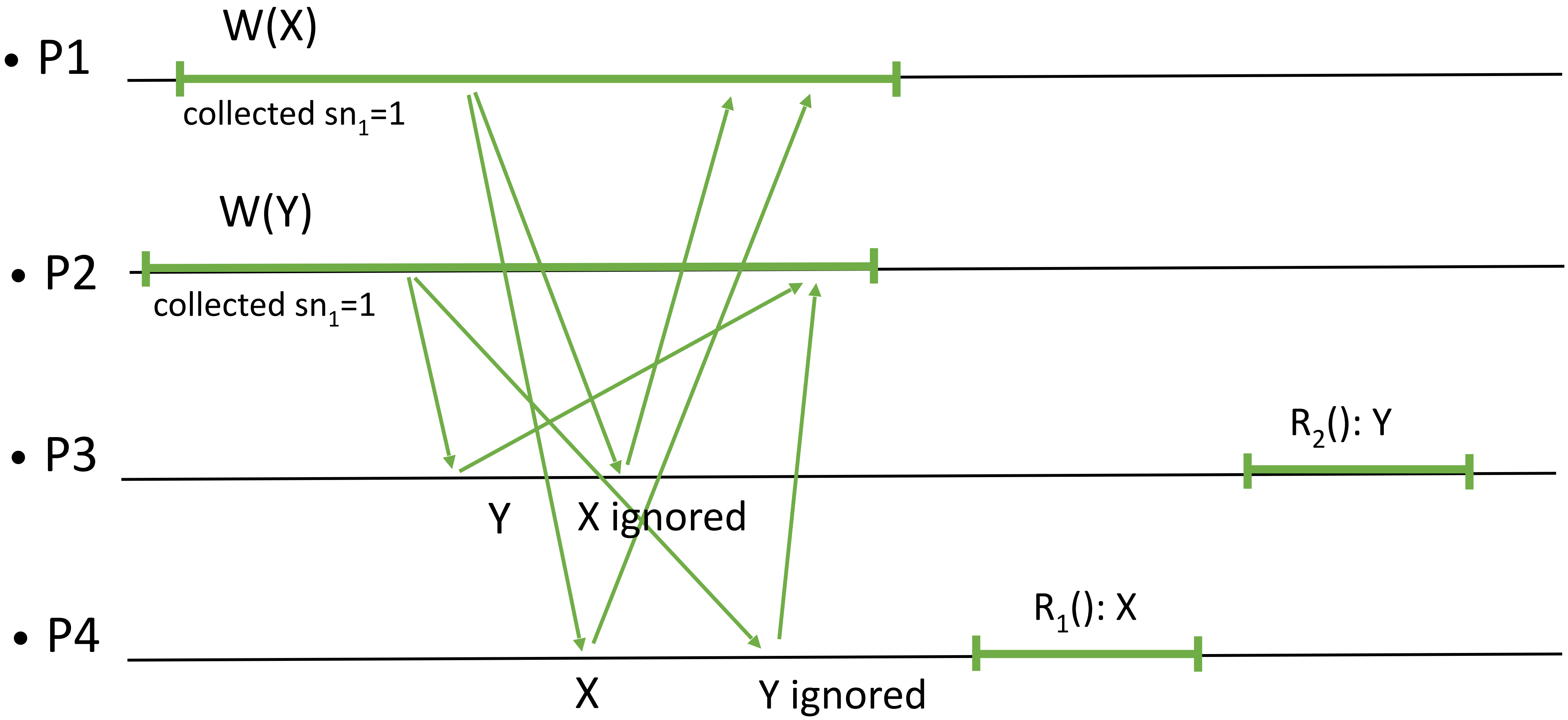
Timestamp not enough?



Timestamp not enough?

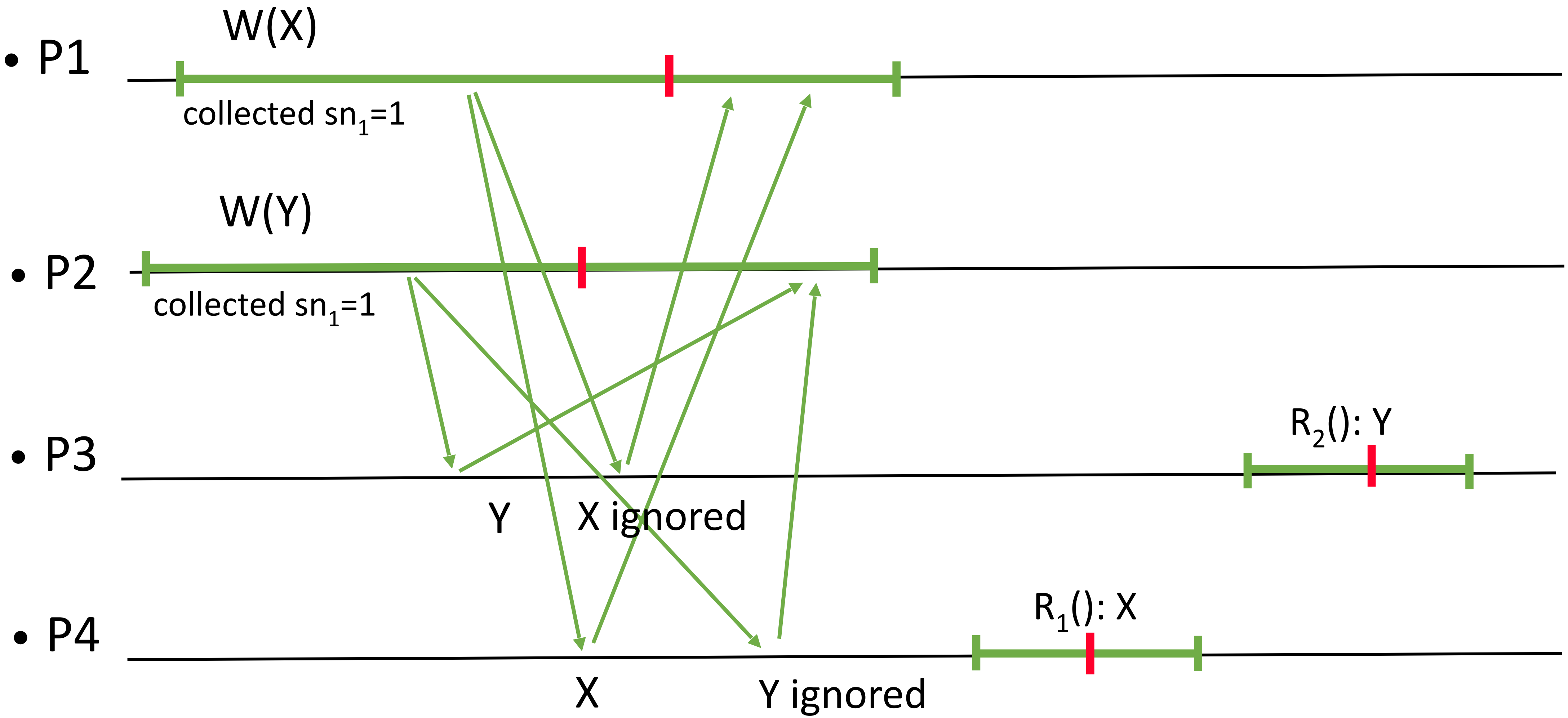


Timestamp not enough?



R_2 should return X but is returning Y. Not linearizable.

Timestamp not enough?



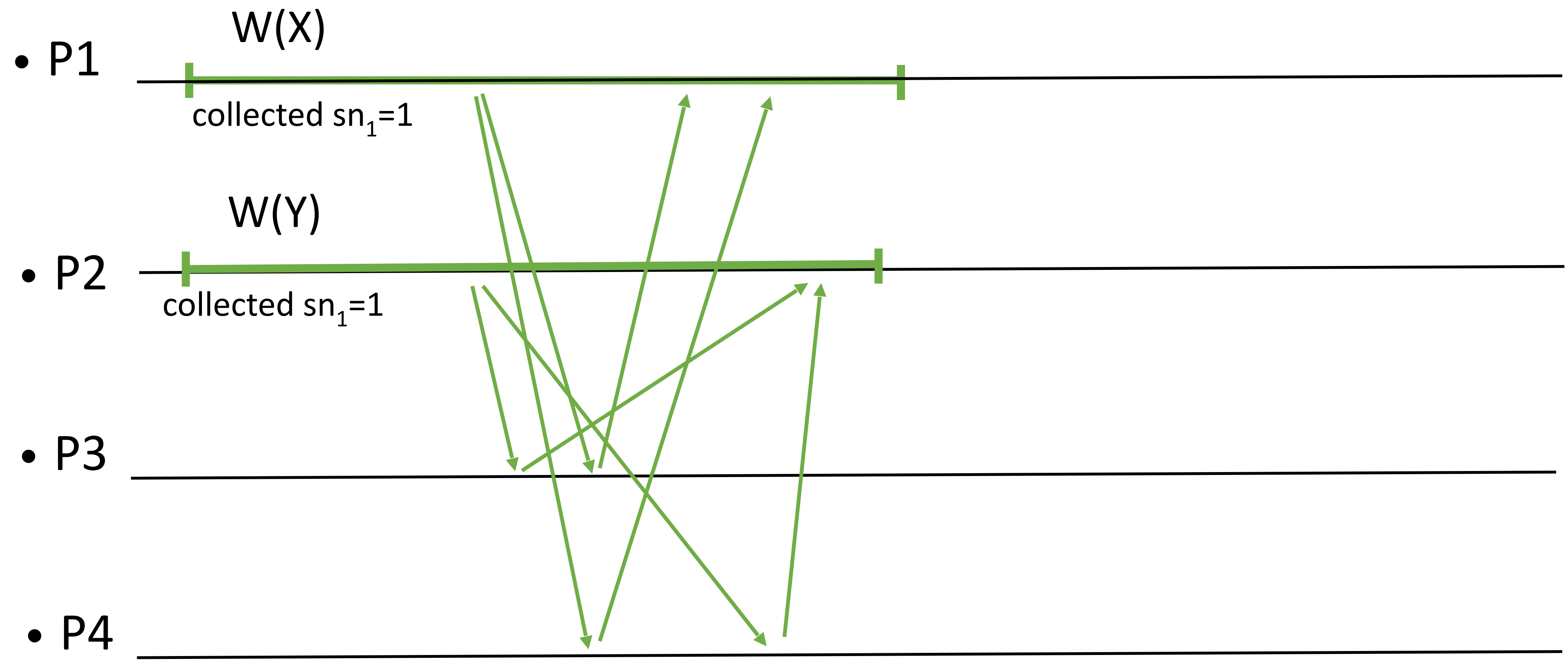
R_2 should return X but is returning Y. Not linearizable.

A fail-stop N-N algorithm

Idea:

- To write, first collect the largest timestamp, and increment it.
- But two writer processes might get the same timestamp at the same time. If their messages are delivered in two different orders to two processes, those processes end up with different values. Then, later reads in them are not linearizable.
- Unique write ids: (ts, pid)
- First timestamps and then a fixed order between processes determine the order.

Unique identifiers



- P1

W(X)

collected sn_1=1

- P2

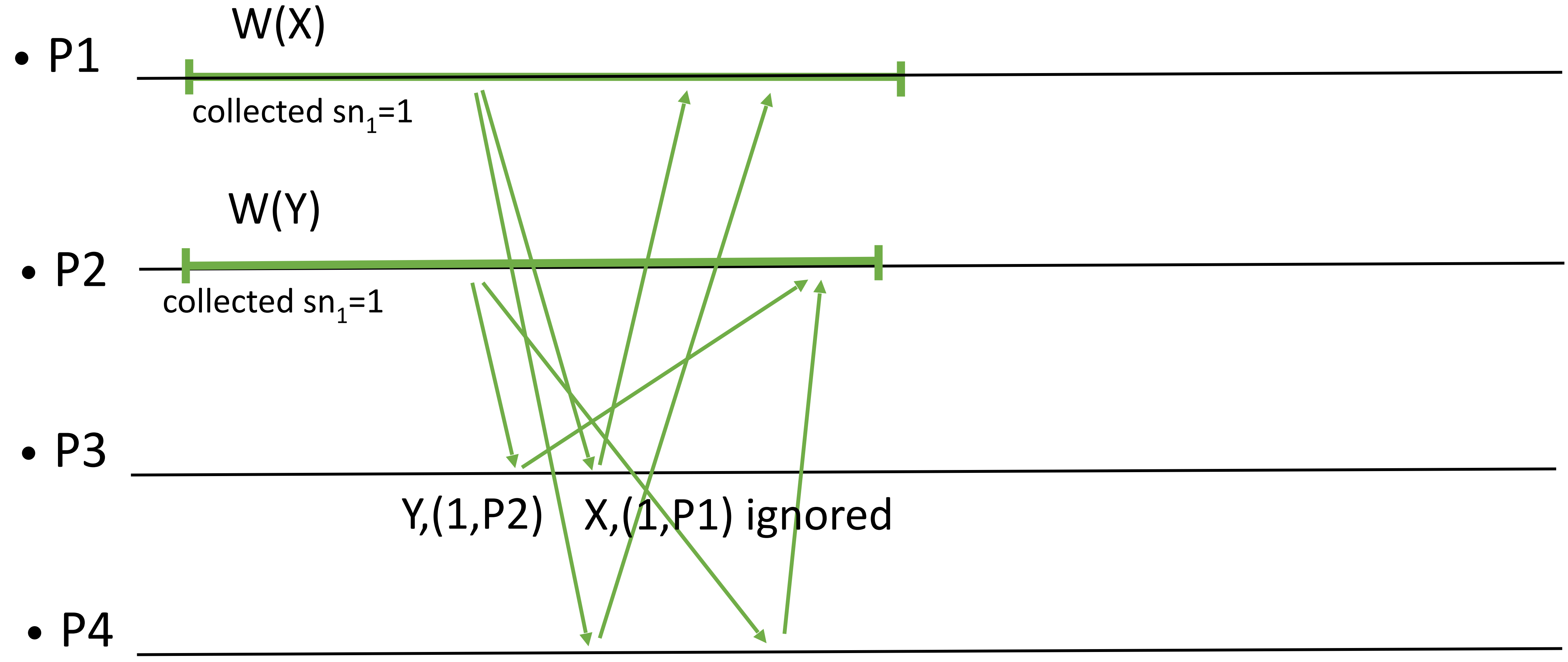
W(Y)

collected sn_1=1

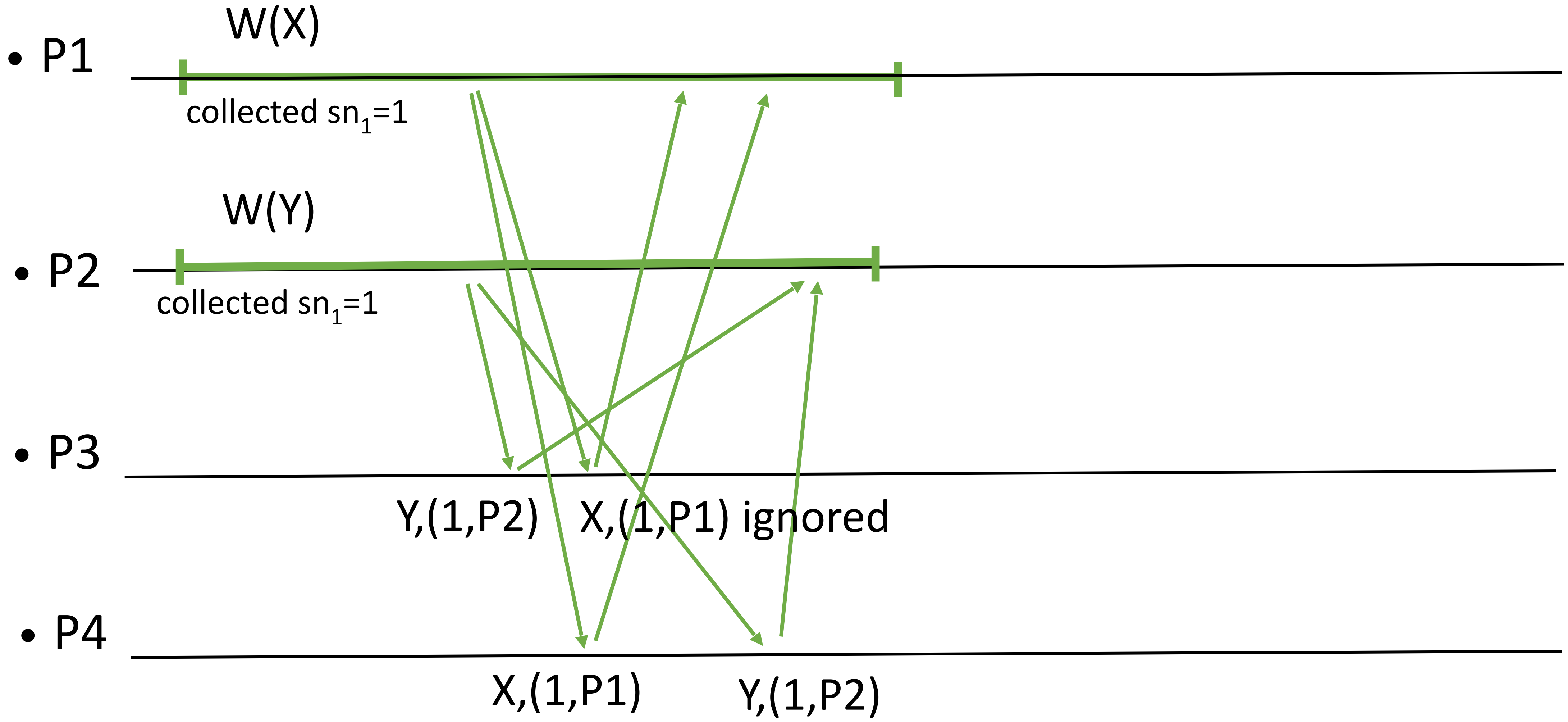
- P3

- P4

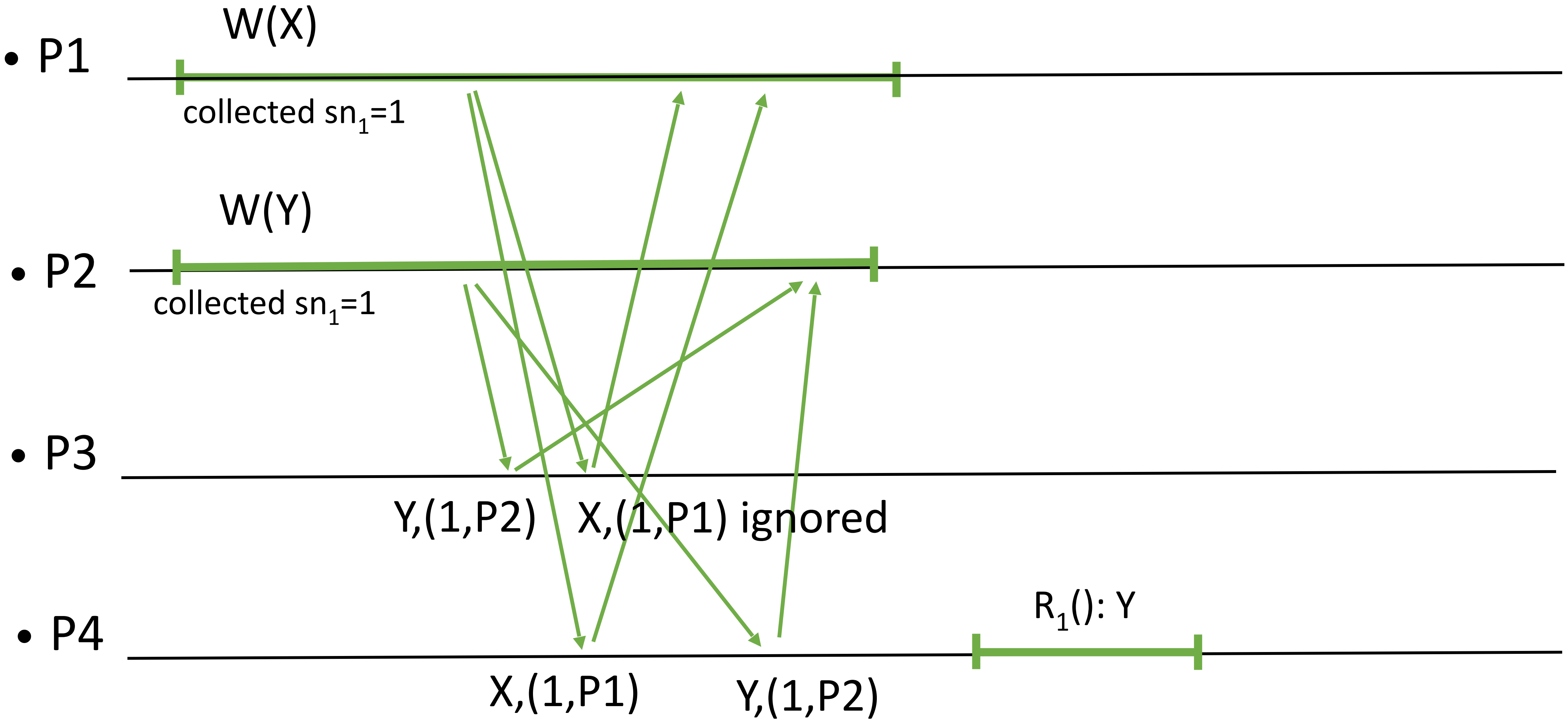
Unique identifiers



Unique identifiers

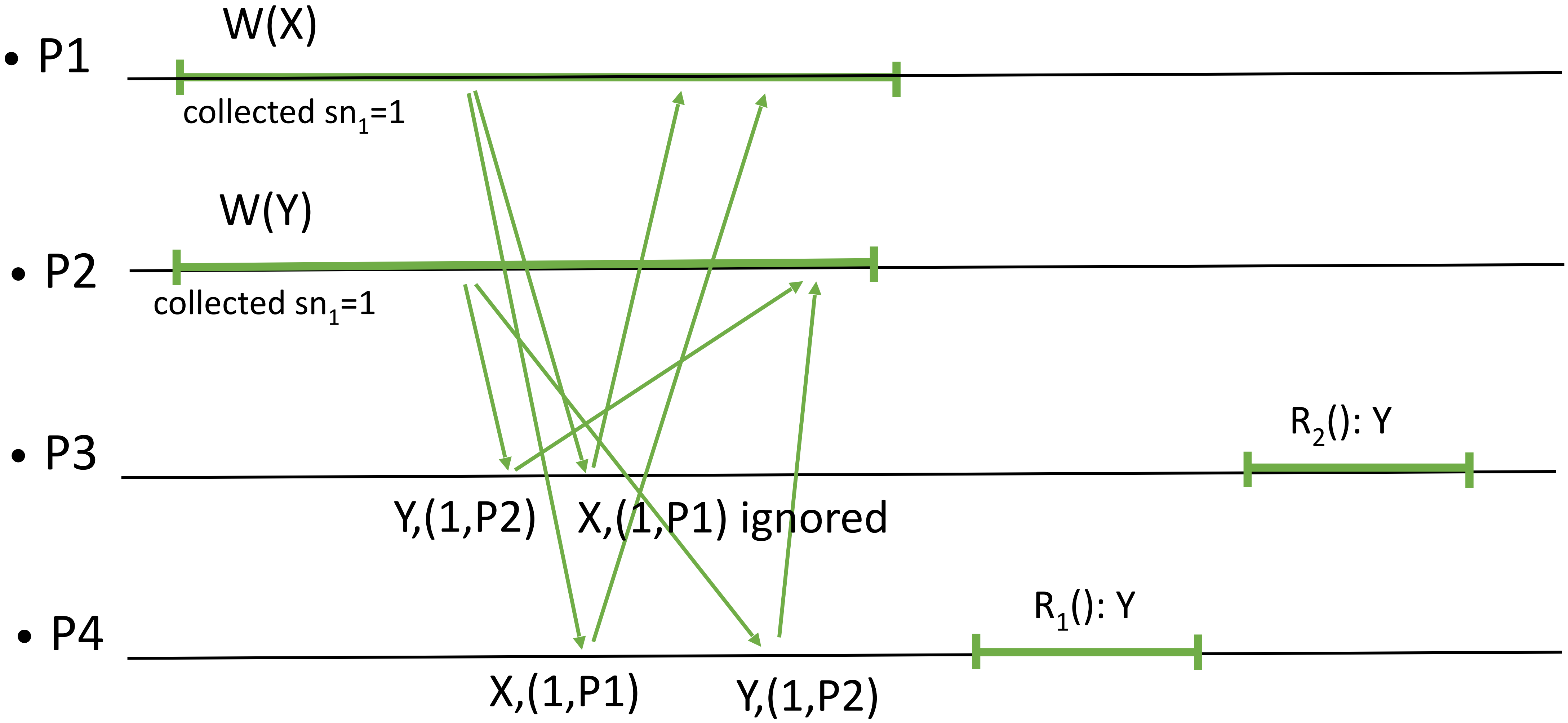


Unique identifiers



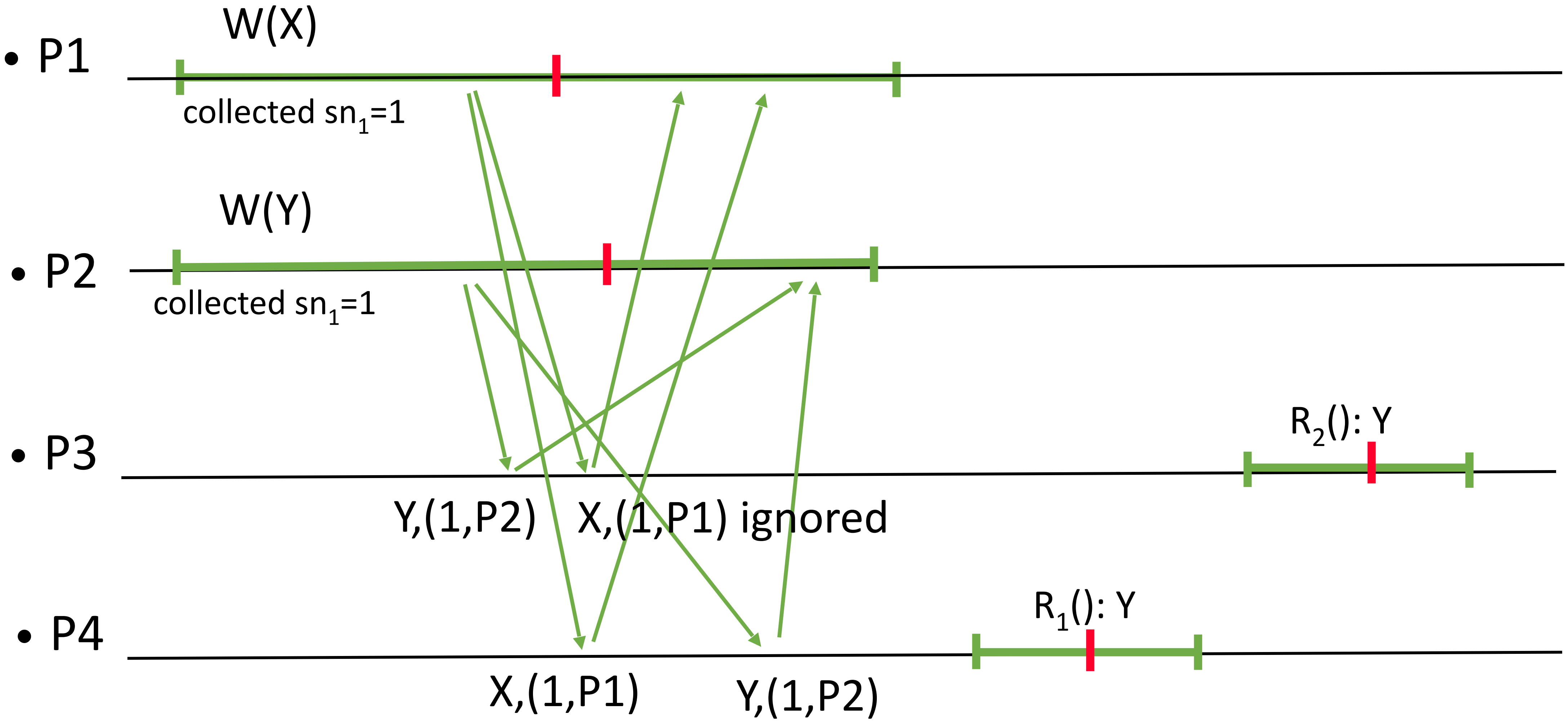
R_1 and R_2 should both return Y. Linearizable.

Unique identifiers



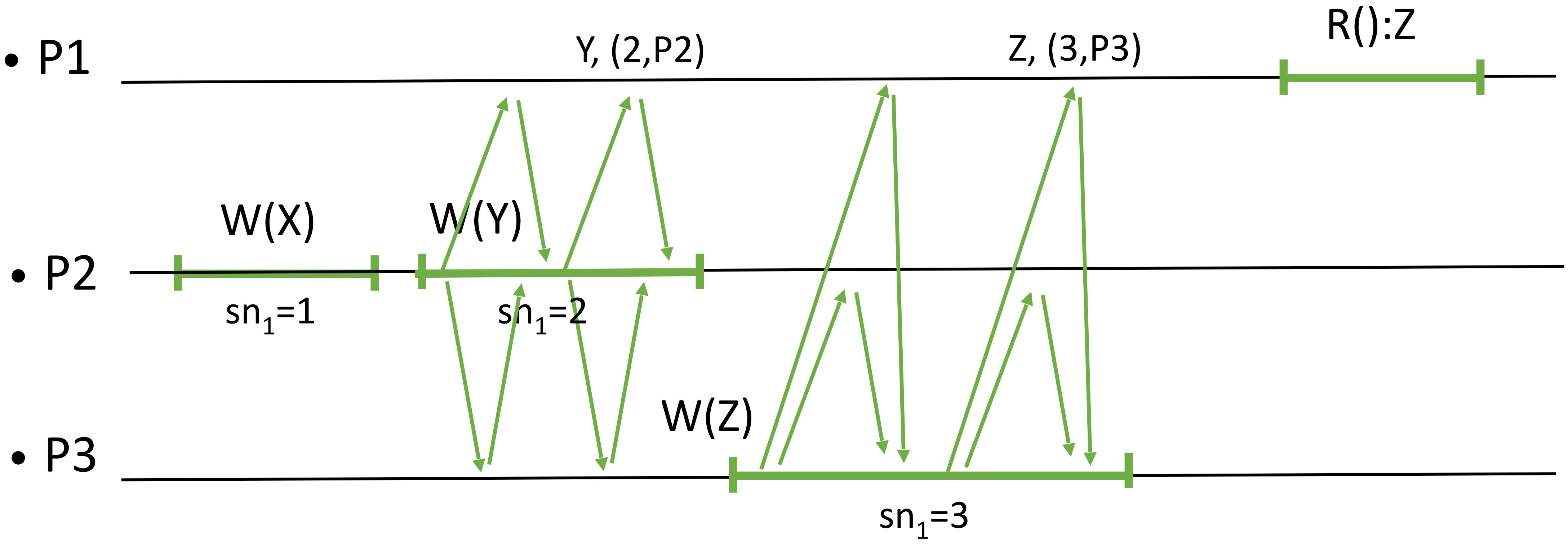
R_1 and R_2 should both return Y. Linearizable.

Unique identifiers



R_1 and R_2 should both return Y. Linearizable.

Unique identifiers



W(Z) collects the largest timestamp 2 and sends updates with timestamp 3.

The Write() Protocol

```
upon Write(v) at $p_i$
  trigger send [W] to all
  foreach $p_j$, wait until either:
    deliver [W,$sn_j$] or
    suspect [$p_j$]
  (sn,id) := (highest $sn_j$ + 1,i)
  trigger send [W,(sn,id),v] to all
  foreach $p_j$, wait until either:
    deliver [W,(sn,id),ack] or
    suspect [$p_j$]
  trigger ok
```

```
At $p_i$ :
  upon deliver [W] from $p_j$
    trigger send [W,$sn_i$] to $p_j$

  upon deliver [W,($sn_j$,$id_j$),v] from $p_j$
    if ($sn_j$,$id_j$) > (sn,id) then
      $v_i$ := v
      (sn,id) := ($sn_j$,$id_j$)
    trigger send [W,($sn_j$,$id_j$),ack] to $p_j$
```

Writes collect the highest timestamp first.

The Read() Protocol

```
upon Read(v) at $p_i$
    trigger send [R] to all
    foreach $p_j$, wait until either:
        deliver [R,($sn_j$,$id_j$),$v_j$] or
        suspect [$p_j$]
    v = $v_j$ with the highest ($sn_j$,$id_j$)
    (sn,id) := highest ($sn_j$,$id_j$)

    trigger send [W,(sn,id),v] to all
    foreach $p_j$, wait until either:
        deliver [W,(sn,id),ack] or
        suspect [$p_j$]

    trigger Ret(v)
```

```
At $p_i$ :
    upon deliver [R] from $p_j$
        trigger send [R,($sn_i$,$id_i$),$v_i$] to $p_j$

    upon deliver [W,($sn_j$,$id_j$),v] from $p_j$
        if ($sn_j$,$id_j$) > (sn,id) then
            $v_i$ := v
            (sn,id) := ($sn_j$,$id_j$)
        trigger send [W,($sn_j$,$id_j$),ack] to $p_j$
```

Reads still try to update other processes with their value before returning it.

Overview of this lecture

1. A 1-1 atomic fail-stop algorithm
2. From regular to atomic
3. A 1-N atomic fail-stop algorithm
4. A N-N atomic fail-stop algorithm
5. **From fail-stop to fail-silent**

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,
  - in addition, the writers first determine the timestamp from a majority.

Parts of slides adopted from R. Guerraoui