DISTRIBUTED REGISTERS II

ATOMIC REGISTERS

CONSISTENCY PROPERTIES

- REGULAR CONSISTENCY: every reads on register x outputs the last value written on x, or the value that someone is concurrently writing on x.
  - We have seen algorithms for (1,N) Regular register with P (fail-stop model) and without P (fail-silent) using quorums and majority of corrects.
- SEQUENTIAL CONSISTENCY: there exists a global ordering of all writes and reads that is consistent with the local order of operations that each process sees.
- ATOMIC CONSISTENCY/(LINEARIZABILITY): Each operation appears to take effect instantaneously at some moment between its start and end.

(1,N) ATOMIC REGISTER: SPECIFICATION

Properties:

Termination. If a correct process invokes an operation, then the operation eventually receives the corresponding confirmation.

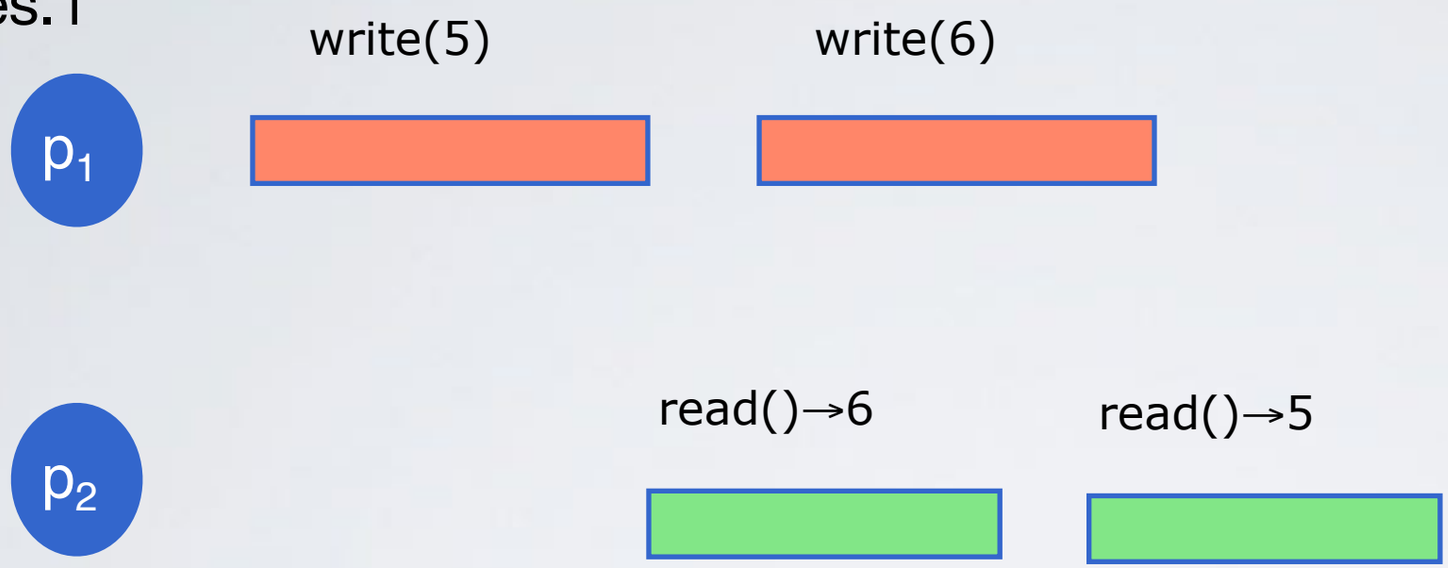
Validity. A read operation returns the last value written or the value concurrently being written.

Ordering. If a read returns v2 after a read that precedes it has returned v1, then v1 has not been written after v2

(Ordering is stated for (1,N) and it is equivalent to the more general definition with Linearization Points).

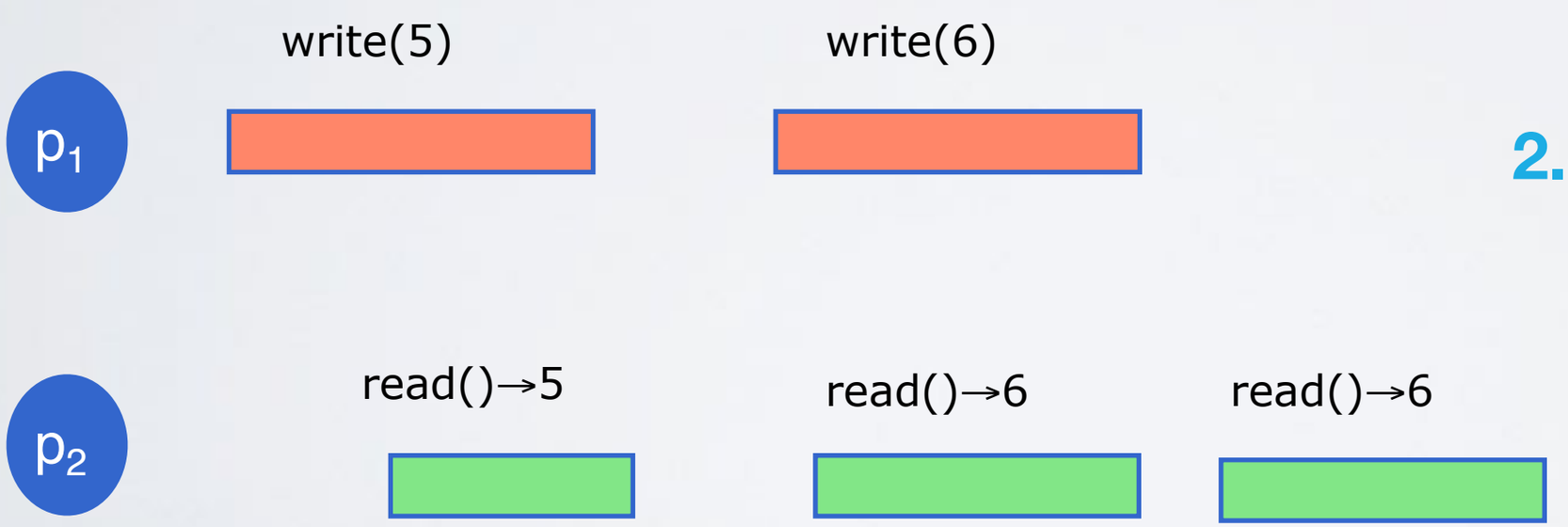
(1,N) ATOMIC REGISTER: SCENARIO

es.1



1. **Regular but not atomic register: Write(5) precedes write(6). But process p_2 read first the value 6 and then the value 5**

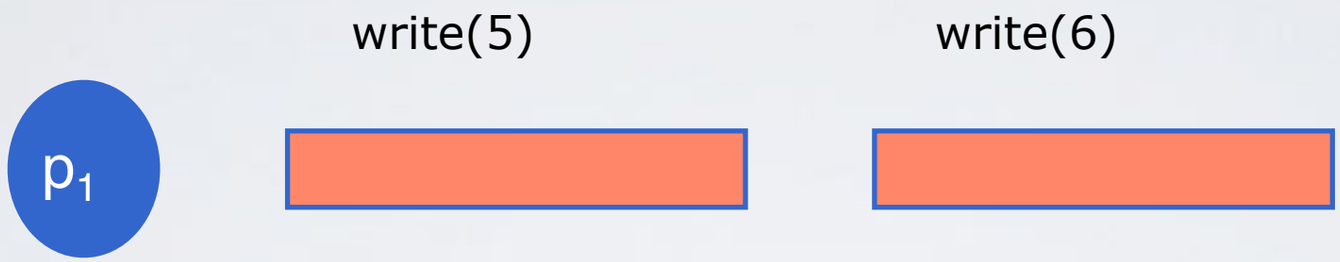
es.2



2. **The register is atomic**

(1,N) ATOMIC REGISTER: SCENARIO

Non atomic register: the precedence relation also refers to read operations issued by different processes

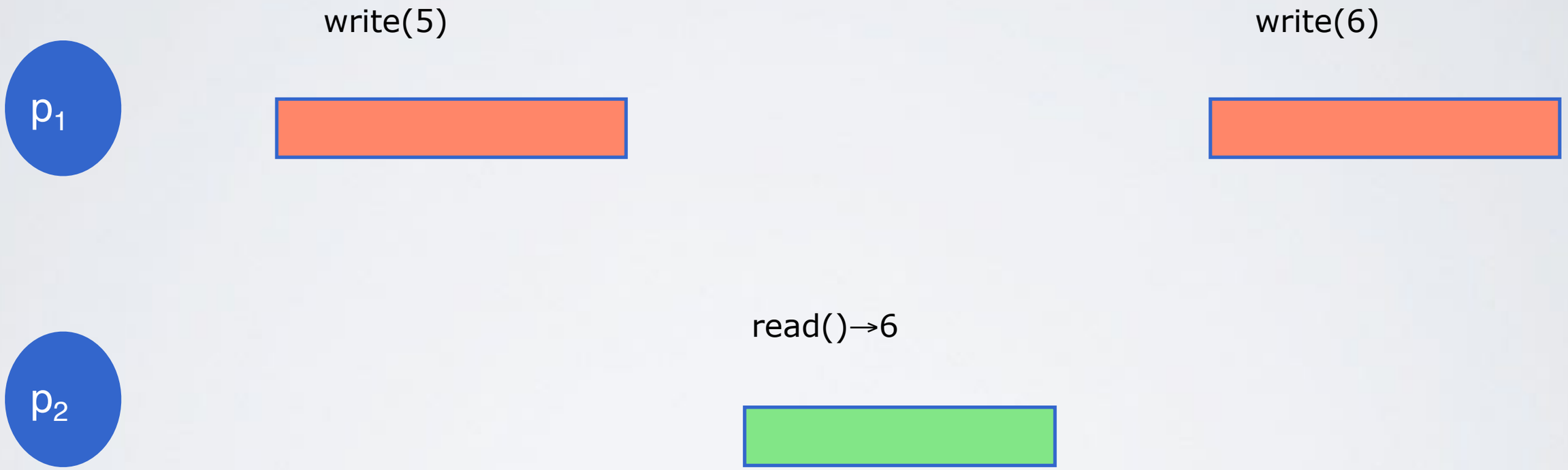


SCENARIO 1

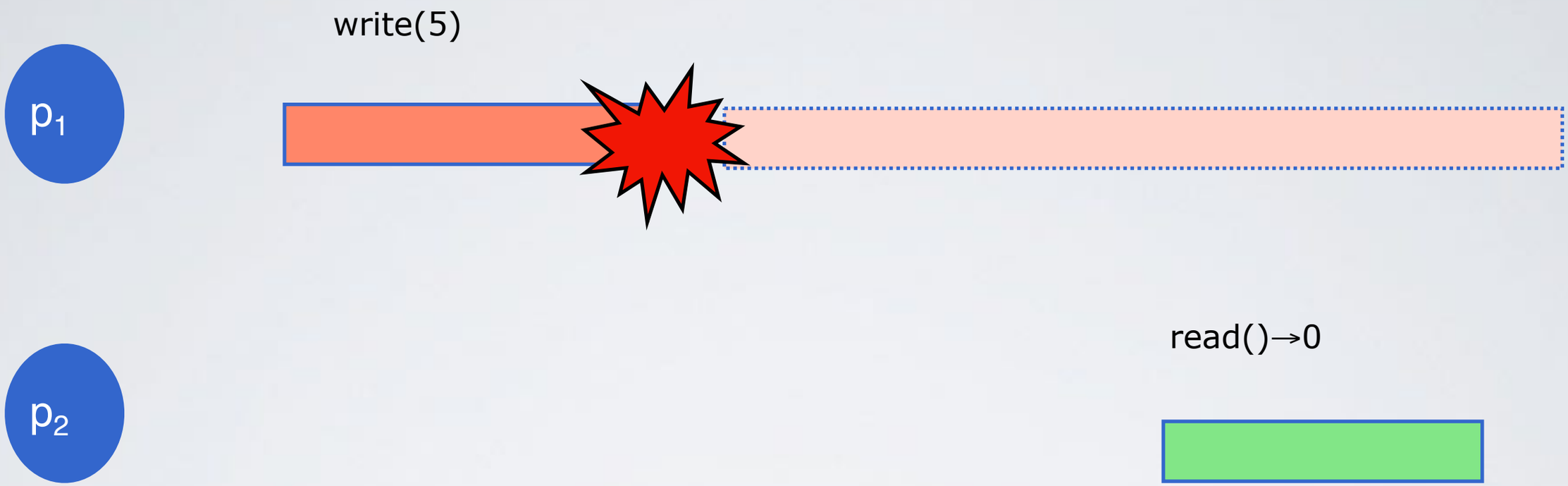
ATOMIC and REGULAR.

SCENARIO 2

NOT ATOMIC and NOT REGULAR, but SEQUENTIAL CONSISTENT

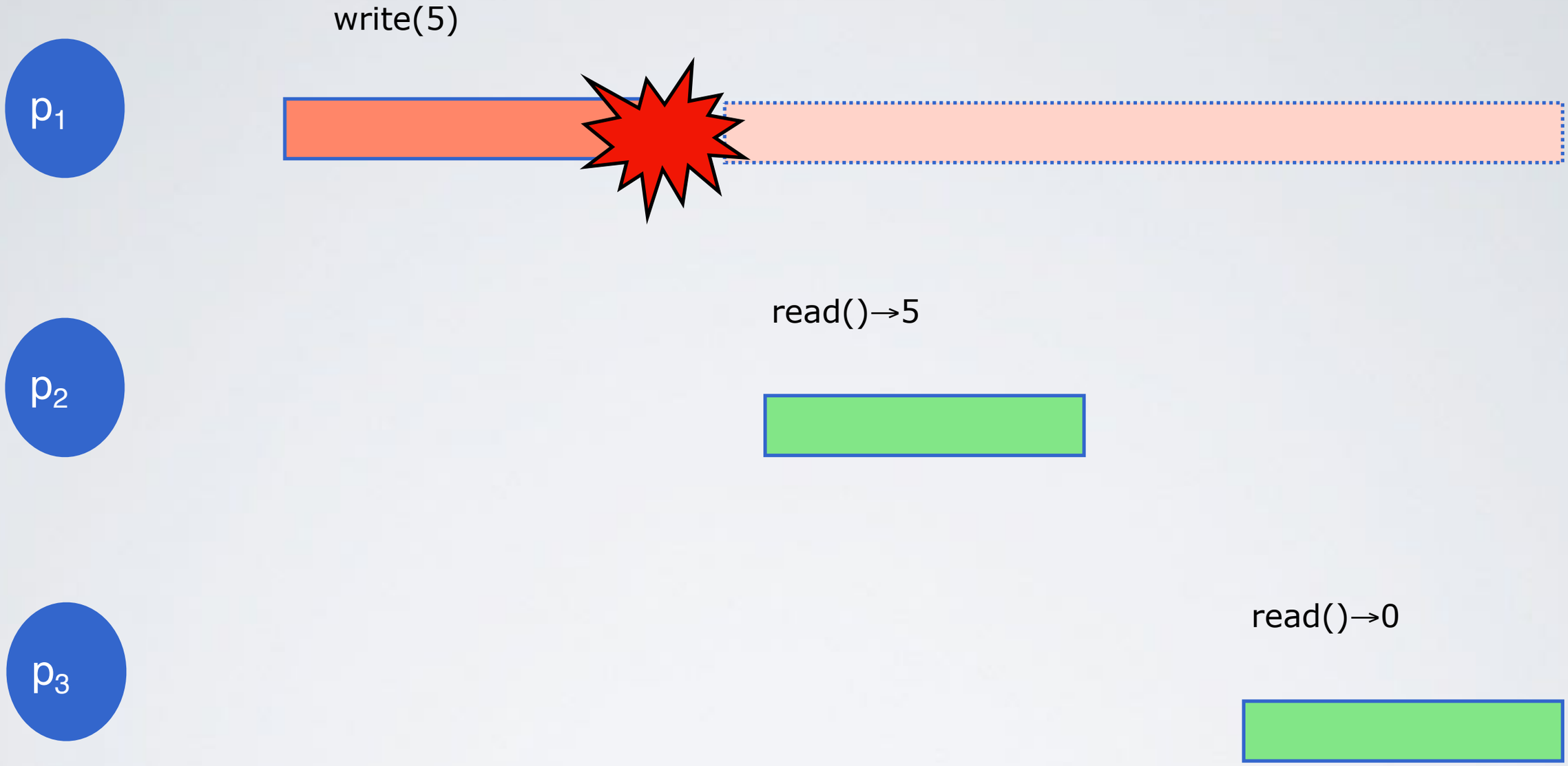


SCENARIO 3



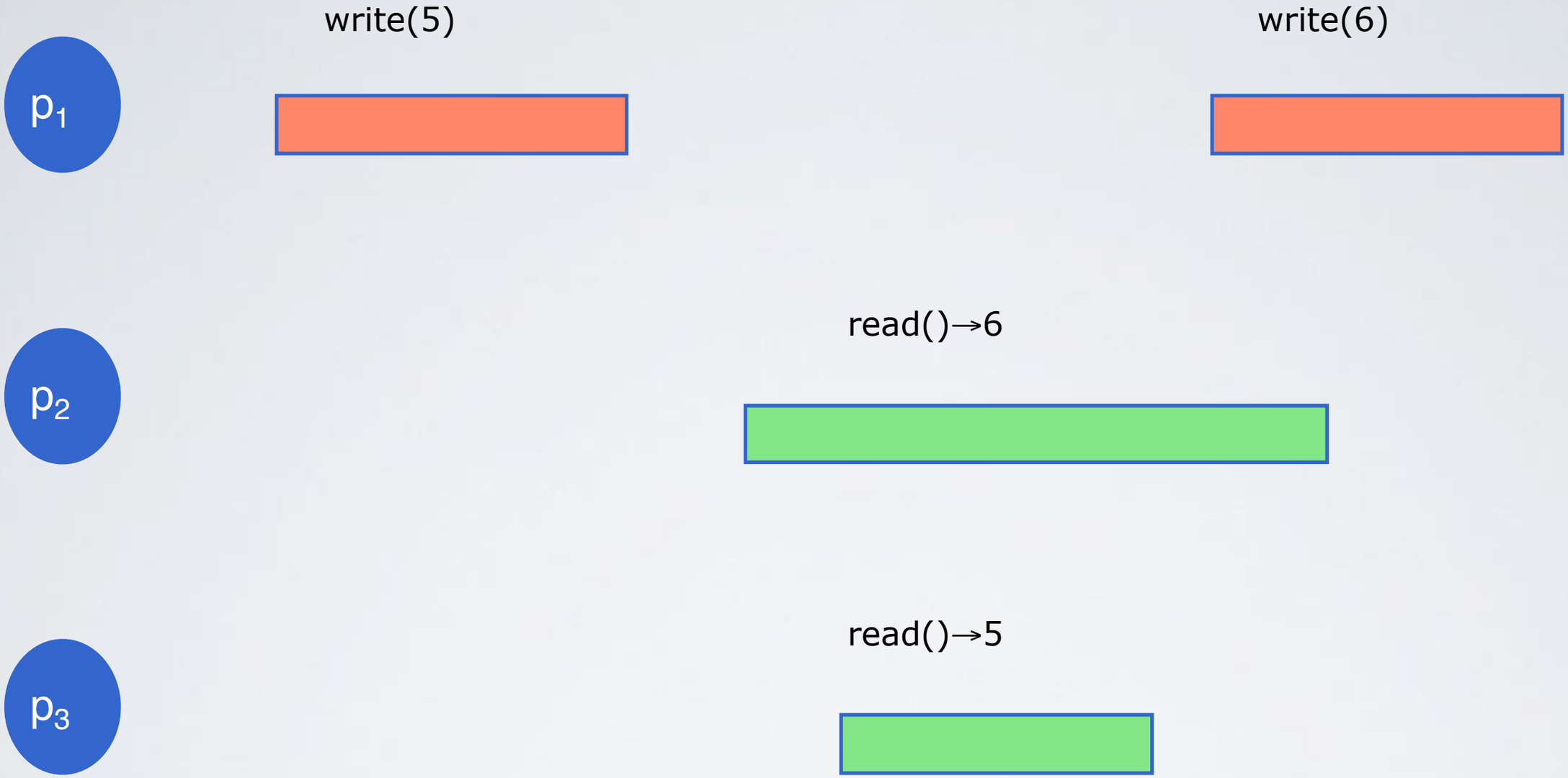
ATOMIC and REGULAR. write(5) executed by p_1 fails. So, it does not complete and it is concurrent with the read by p_2. Validity is respected.

SCENARIO 4



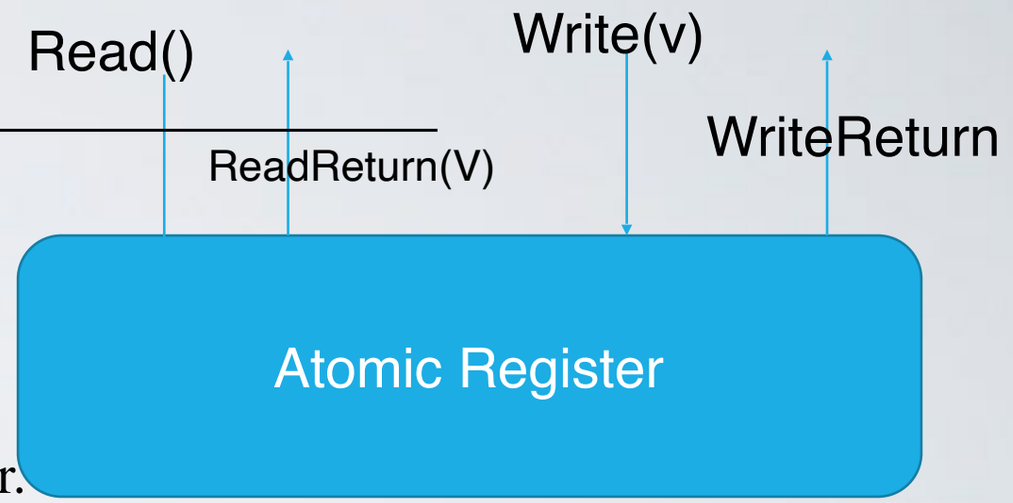
REGULAR but non ATOMIC. The ordering property is violated.

SCENARIO 5



ATOMIC and REGULAR

(1 ,N) ATOMIC REGISTER: INTERFACE



Module 4.2: Interface and properties of a $(1, N)$ atomic register

Module:

Name: $(1, N)$-AtomicRegister, **instance** *onar*.

Events:

Request: $\langle$ *onar*, *Read* $\rangle$: Invokes a read operation on the register.

Request: $\langle$ *onar*, *Write* $\mid v \rangle$: Invokes a write operation with value v on the register.

Indication: $\langle$ *onar*, *ReadReturn* $\mid v \rangle$: Completes a read operation on the register with return value v.

Indication: $\langle$ *onar*, *WriteReturn* $\rangle$: Completes a write operation on the register.

Properties:

ONAR1–ONAR2: Same as properties ONRR1–ONRR2 of a $(1, N)$ regular register (Module 4.1).

ONAR3: *Ordering:* If a read returns a value v and a subsequent read returns a value w, then the write of w does not precede the write of v.

ALGORITHMS

We will show algorithms to implement (1,N) Atomic Register using (1,N) Regular.

- WE WILL NOT USE MESSAGES TO COMMUNICATE, JUST Regular Registers

We will show algorithms to implement (1,N) Atomic Register using directly message passing.

- We will only use messages to communicate

(1,N) ATOMIC REGISTER USING REGULAR REGISTER

(1,N) ATOMIC REGISTER -

The algorithm consists of two phases

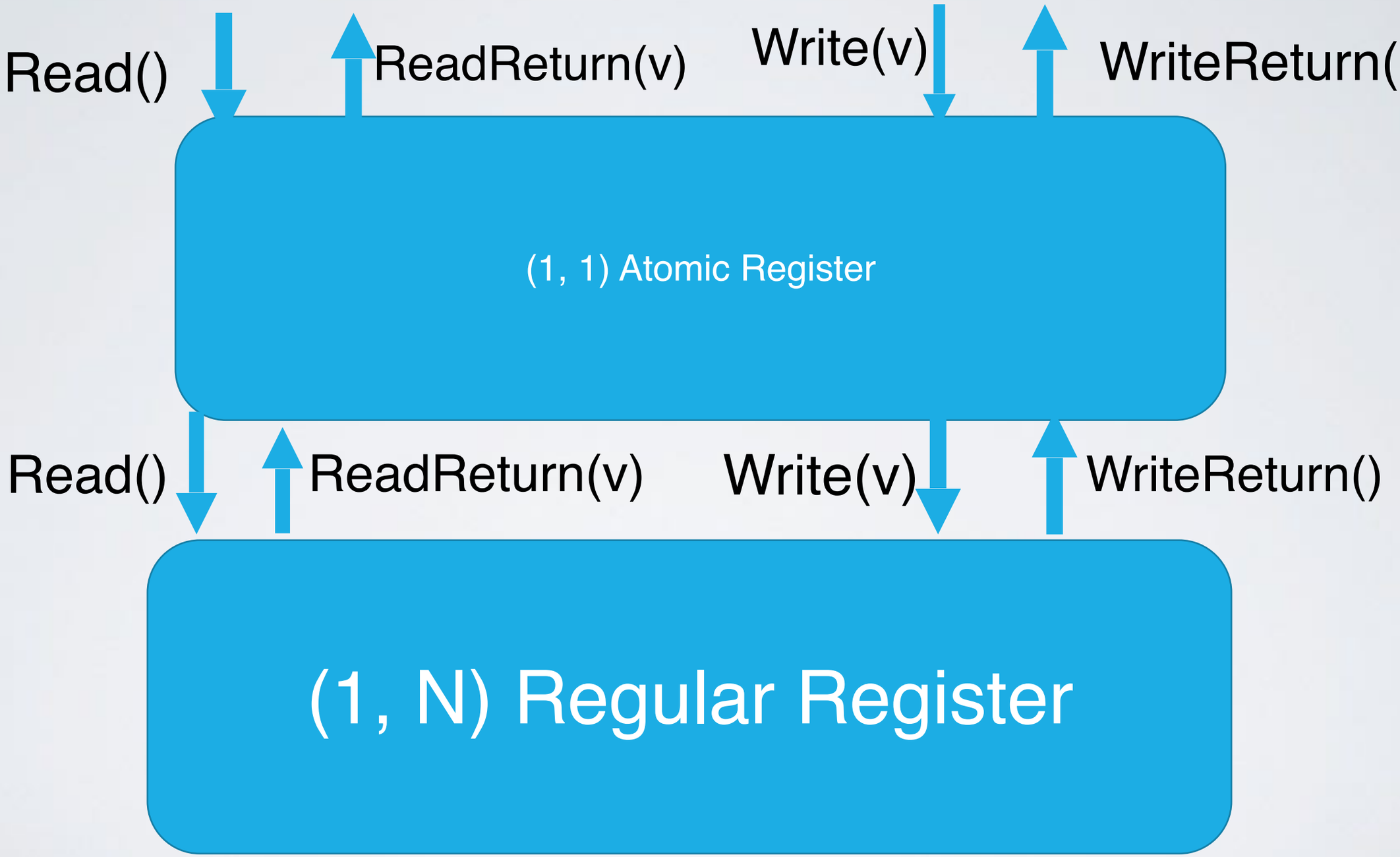
PHASE 1. We use a (1,N) regular register to build a (1,1) atomic register

PHASE 2. We use a set of (1,1) atomic registers to build a (1,N) atomic register

NOTATION:

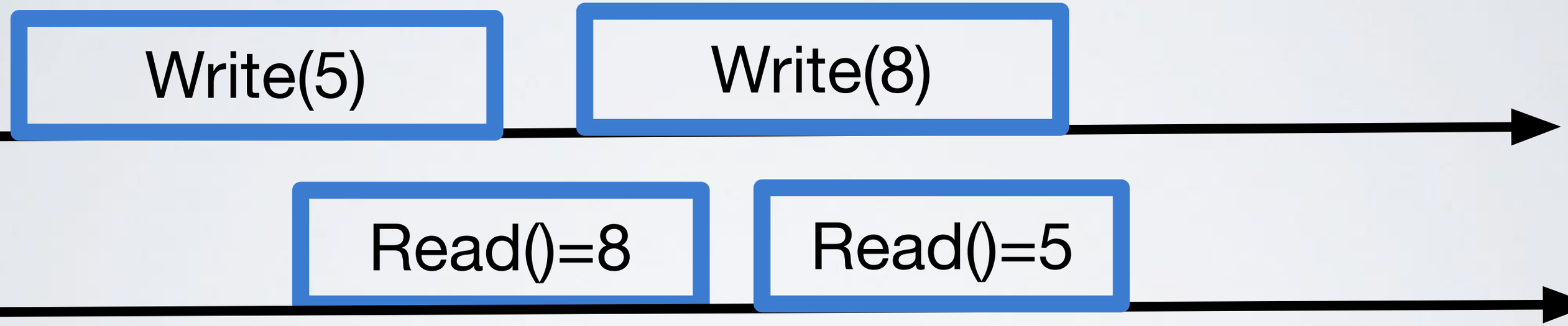
Hereafter, **RR** and **AR**, will be sometimes used to respectively denote Regular Register and Atomic Register

PHASE 1 - FROM (1,N)-RR TO (1,1)-AR



PHASE 1 - FROM (1,N)-RR TO (1,1)-AR

IDEA 1: We have just one writer and one reader, what is the pathological case with a RR that is not acceptable in AR?



PHASE 1 - FROM (1,N)-RR TO (1,1)-AR

IDEA:

- p_1 is the writer and p_2 is the reader of the (1,1) atomic register, we aim to implement
- We use a (1,N) regular register where p_1 is the writer and p_2 is the reader
- Each write operation on the atomic register writes the pair *(value, timestamp)* into the underlying regular register
- The reader tracks the timestamp of previously read values to avoid to read something old

PHASE 1 - FROM (1,N)-RR TO (1,1)-AR

Algorithm 4.3: From $(1, N)$ Regular to $(1, 1)$ Atomic Registers

Implements:

$(1, 1)$-AtomicRegister, **instance** *ooar*.

Uses:

$(1, N)$-RegularRegister, **instance** *onrr*.

upon event $\langle$ *ooar*, *Init* $\rangle$ **do**

$(ts, val) := (0, \perp)$;

$wts := 0$;

upon event $\langle$ *ooar*, *Write* | v $\rangle$ **do**

$wts := wts + 1$;

trigger $\langle$ *onrr*, *Write* | (wts, v) $\rangle$;

upon event $\langle$ *onrr*, *WriteReturn* $\rangle$ **do**

trigger $\langle$ *ooar*, *WriteReturn* $\rangle$;

upon event $\langle$ *ooar*, *Read* $\rangle$ **do**

trigger $\langle$ *onrr*, *Read* $\rangle$;

upon event $\langle$ *onrr*, *ReadReturn* | (ts', v') $\rangle$ **do**

if $ts' > ts$ **then**

$(ts, val) := (ts', v')$;

trigger $\langle$ *ooar*, *ReadReturn* | val $\rangle$;

PHASE 1 - FROM (1,N)-RR TO (1,1)-AR

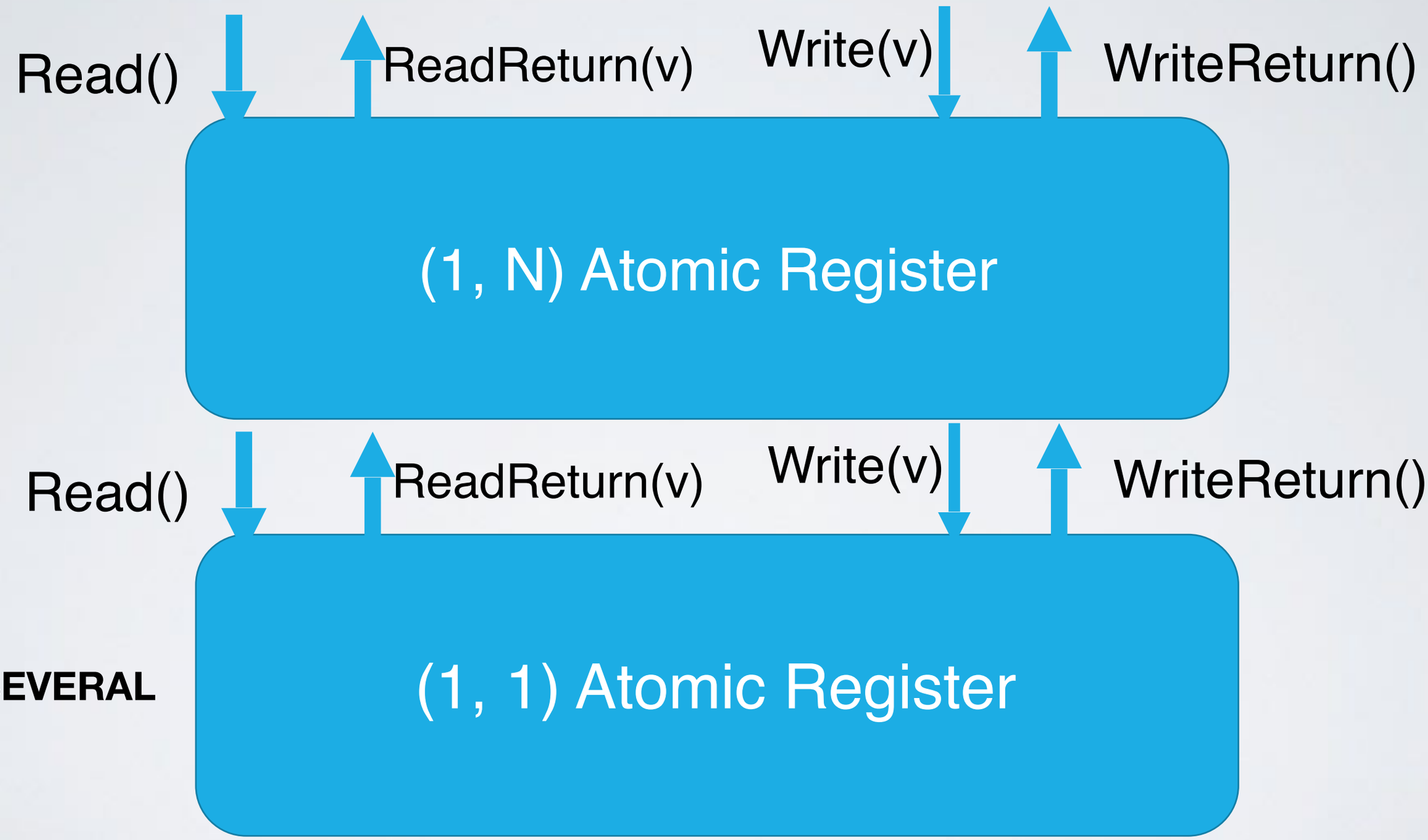
Correctness:

- **Termination** – from the termination property of the regular register
- **Validity** – from the validity property of the regular register
- **Ordering** – from the validity property and from the fact that the read tracks the last value read and its timestamp. A read operation always returns a value with a timestamp greater or equal to the one of the previously read value

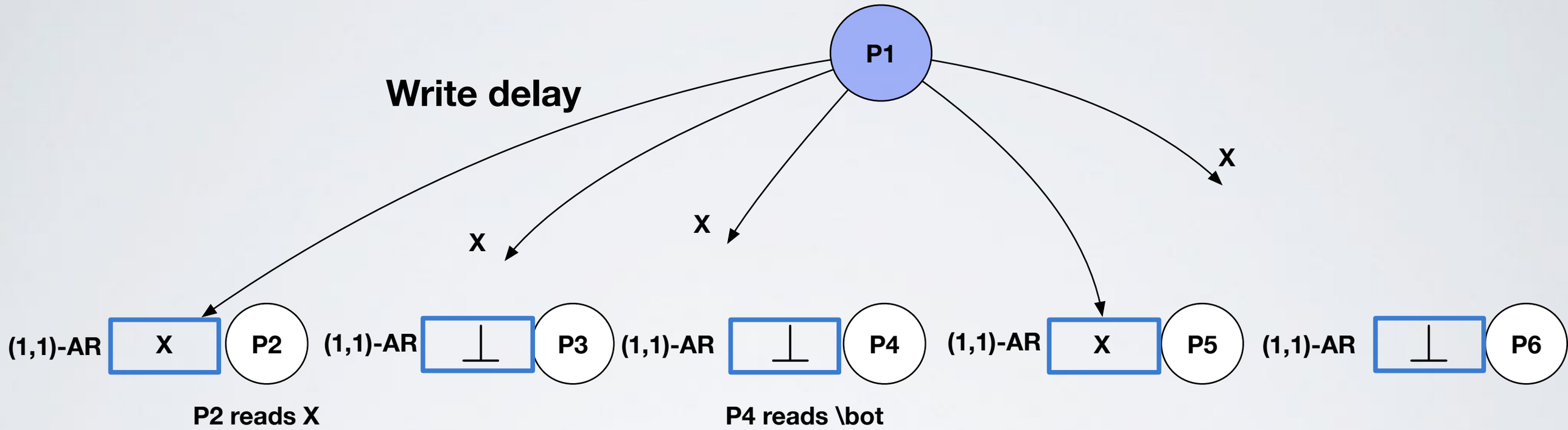
Performance:

- **Write** – Each write operation performs a write on a (1,N) regular register
- **Read** - Each read operation performs a read on a (1,N) regular register

PHASE 2 - FROM (1,1)-AR TO (1,N)-AR



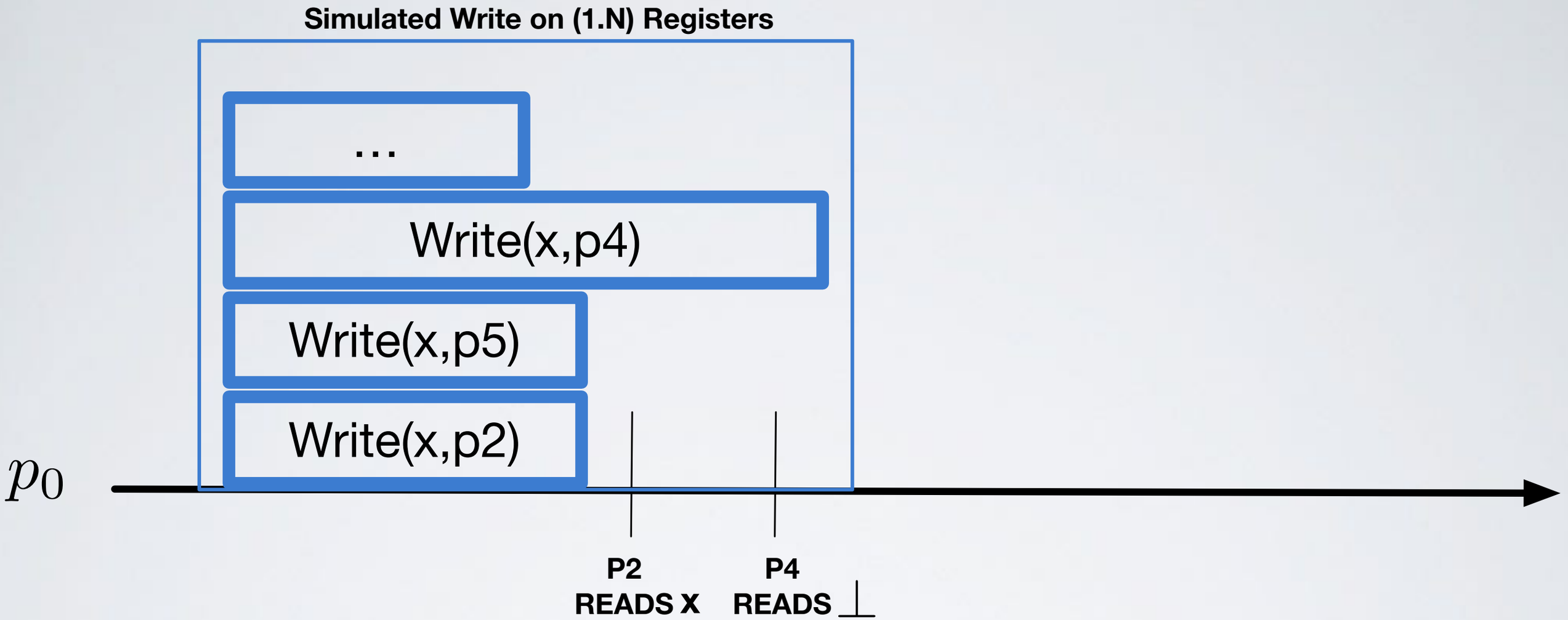
PHASE 2 - FROM (1,1)-AR TO (1,N)-AR



Each one has a (1,1)-register on which P1 writes.

Problem: multiple writes on different registers are not instantaneous, so a write can appear on P2 but not on P4.

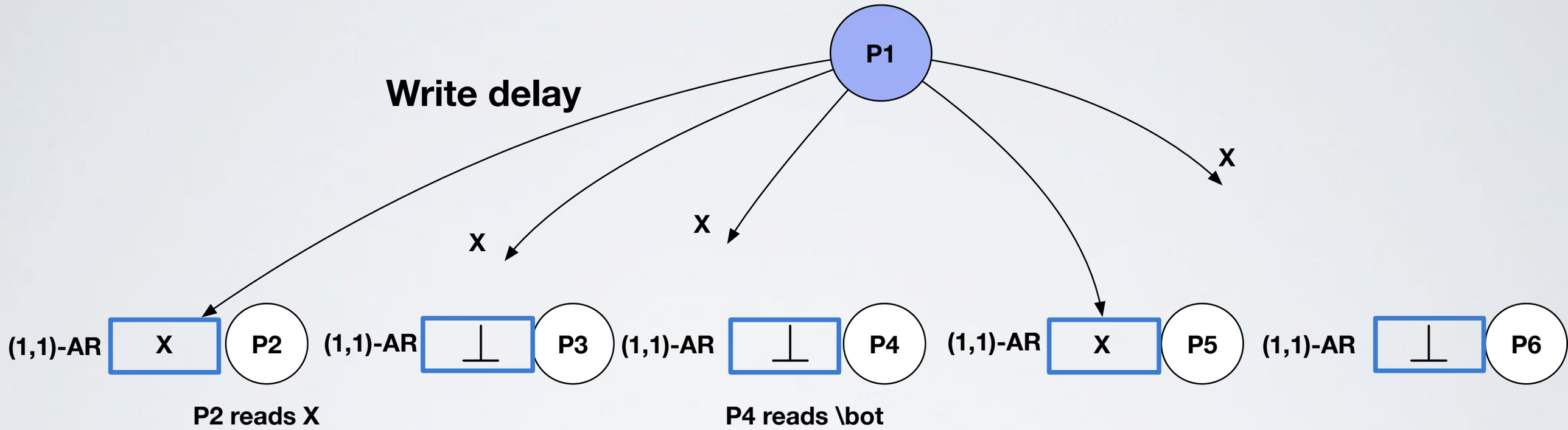
PHASE 2 - FROM (1,1)-AR TO (1,N)-AR



Each one has a (1,1)-register on which P1 writes.

Problem: multiple writes on different registers are not instantaneous, so a write can appear on P2 but not on P4. This does not satisfy an atomic registers

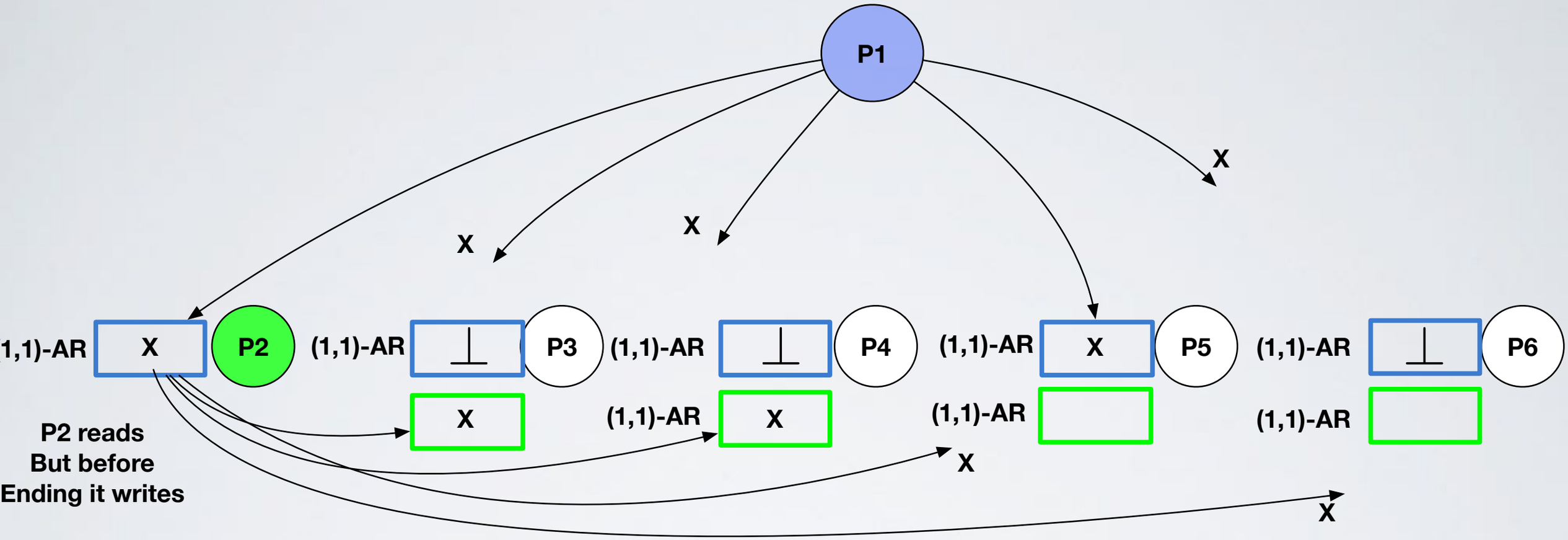
PHASE 2 - FROM (1,1)-AR TO (1,N)-AR



Problem: multiple writes on different registers are not instantaneous, so a write can appear on P2 but not on P4.

How to fix it?

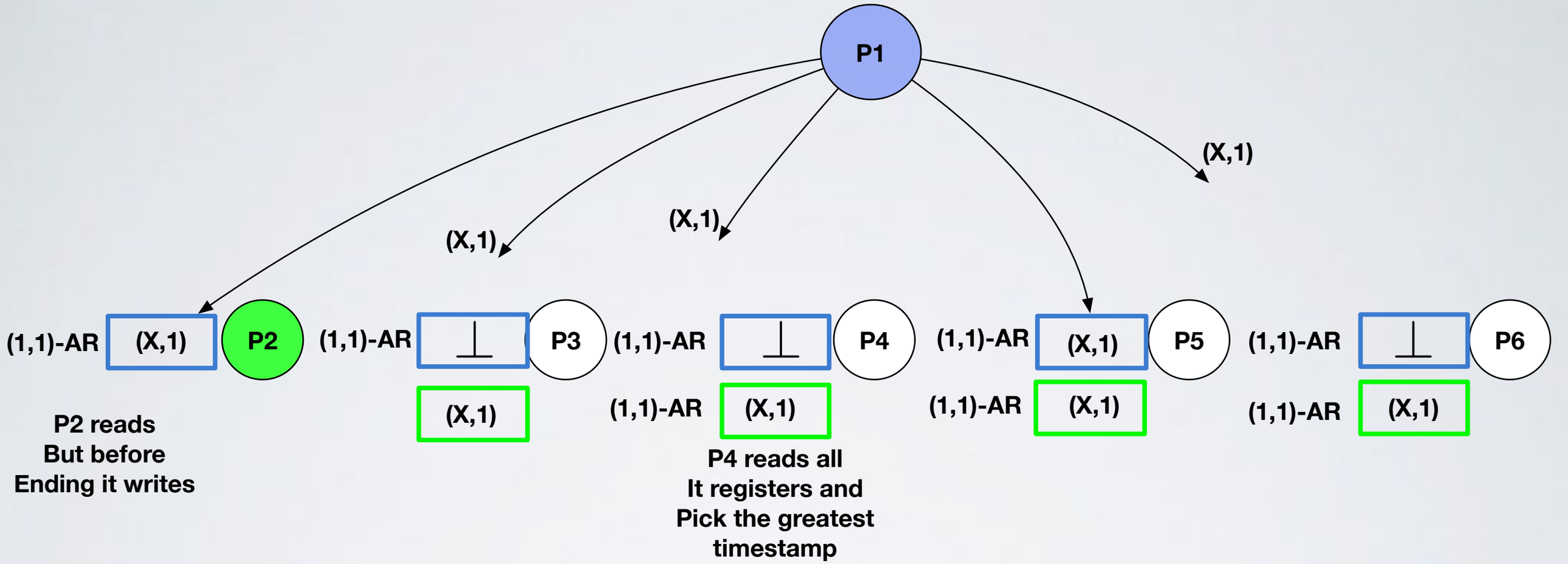
PHASE 2 - FROM (1,1)-AR TO (1,N)-AR



Problem: multiple writes on different registers are not instantaneous, so a write can appear on P2 but not on P4.

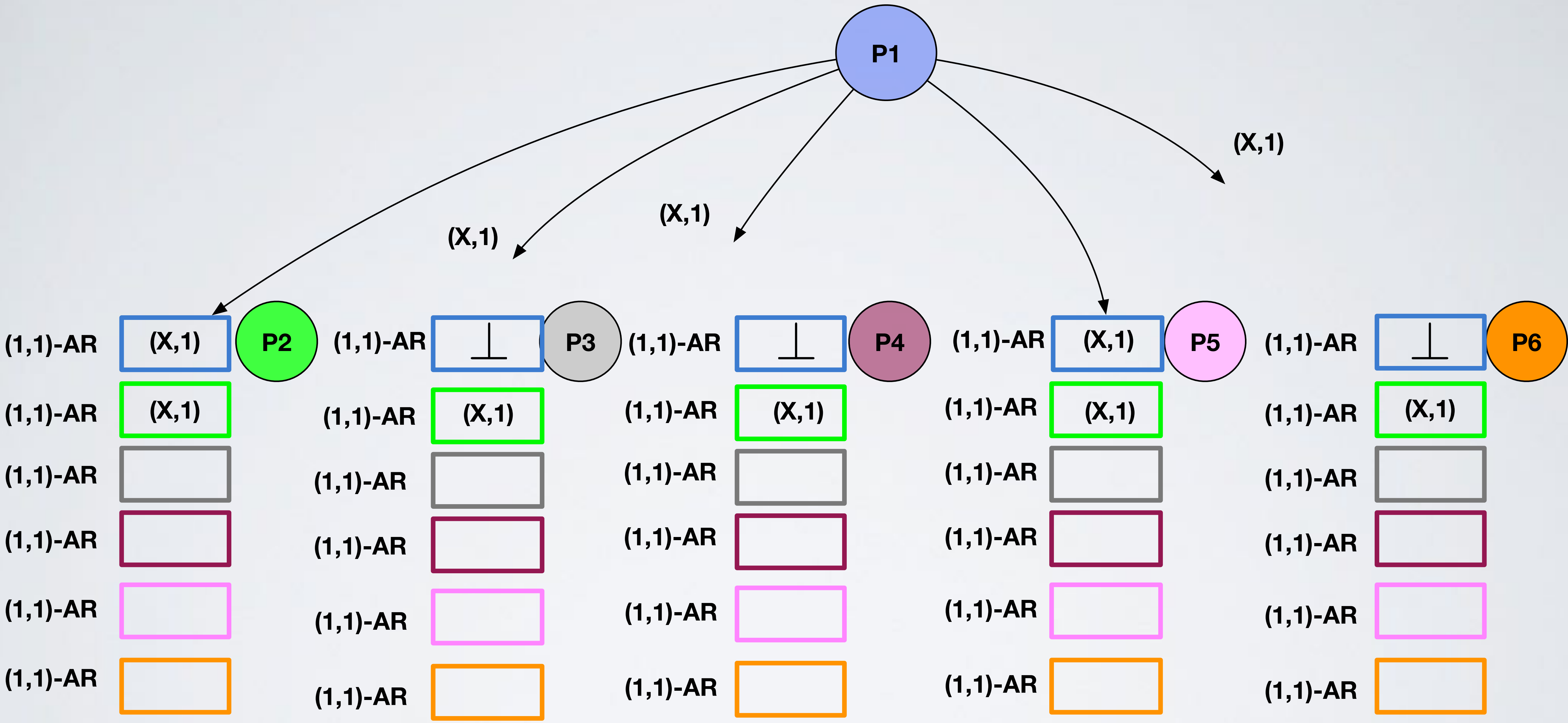
How to fix it? When P2 reads, before completing its operation writes the Value X on the other processes using another register

PHASE 2 - FROM (1,1)-AR TO (1,N)-AR



We need timestamp to decide the last value. A process reads the incoming registers In which P1 and P2 can write.

PHASE 2 - FROM (1,1)-AR TO (1,N)-AR



The same case could happen for other processes, so we need a matrix of registers.

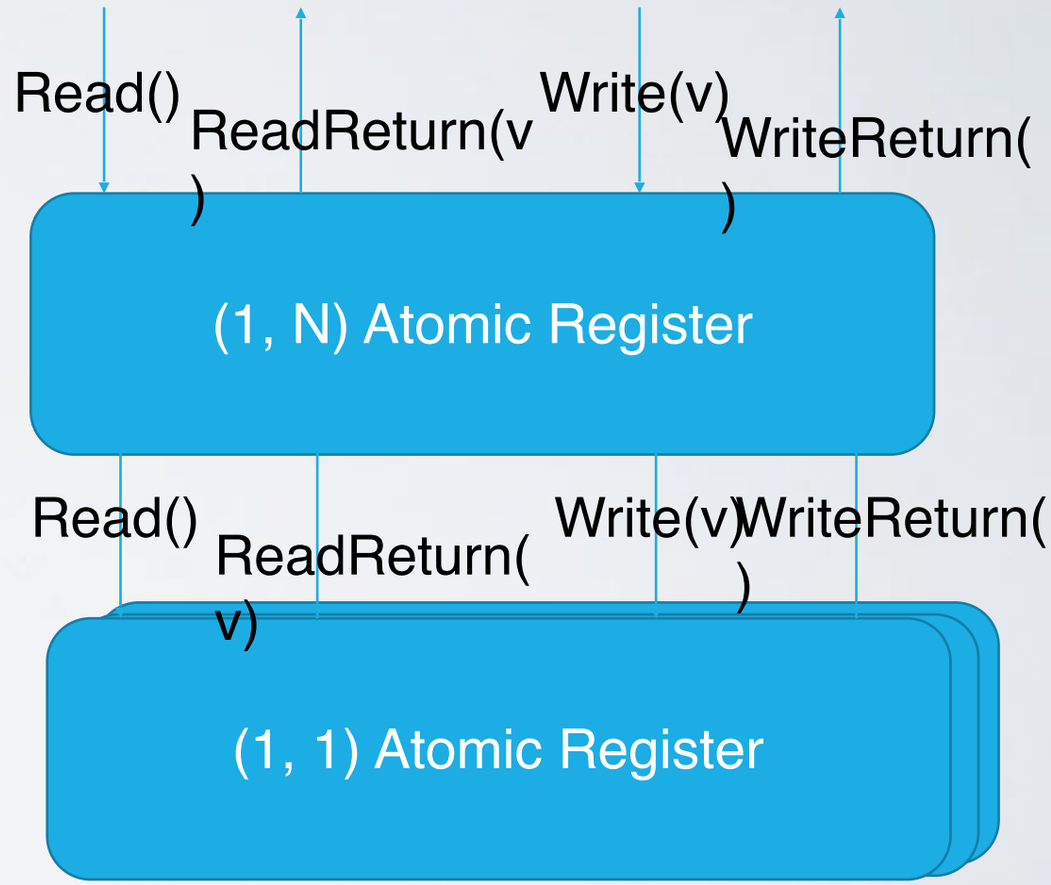
PHASE 2 - FROM (1,1)-AR TO (1,N)-AR

(1,1) atomic → (1,N) atomic

- R_A: N (1,1) registers that connect the writer to the readers
- R_B: N^2 (1,1) registers that connect readers

Idea:

- Write writes val in registers R_A
- Reader
  - reads from 1 R_A register and from N R_B registers
  - chooses the value val with the largest timestamp
  - writes val in N R_B registers
  - returns val



PHASE 2 - FROM (1,1)-AR TO (1,N)-AR

upon event $\langle$ *onar*, *Init* $\rangle$ **do**

$ts := 0$;

$acks := 0$;

$writing :=$ FALSE;

$readval := \perp$;

$readlist := [\perp]^N$;

forall $q \in \Pi, r \in \Pi$ **do**

Initialize a new instance $ooar.q.r$ of $(1, 1)$-AtomicRegister with writer r and reader q;

OOAR.X.Y is a single register in a matrix
X is the reader
And Y is the writer

PHASE 2

upon event $\langle$ *onar*, *Write* | v $\rangle$ **do**
    $ts := ts + 1$;
    writing := TRUE;
    forall $q \in \Pi$ **do**
        trigger $\langle$ *ooar.q.self*, *Write* | (ts, v) $\rangle$;

upon event $\langle$ *ooar.q.self*, *WriteReturn* $\rangle$ **do**
    acks := *acks* + 1;
    if *acks* = N **then**
        acks := 0;
        if *writing* = TRUE **then**
            trigger $\langle$ *onar*, *WriteReturn* $\rangle$;
            writing := FALSE;
        else
            trigger $\langle$ *onar*, *ReadReturn* | *readval* $\rangle$;

upon event $\langle$ *onar*, *Read* $\rangle$ **do**
    forall $r \in \Pi$ **do**
        trigger $\langle$ *ooar.self.r*, *Read* $\rangle$;

upon event $\langle$ *ooar.self.r*, *ReadReturn* | (ts', v') $\rangle$ **do**
    readlist[r] := (ts', v');
    if #(*readlist*) = N **then**
        (*maxts*, *readval*) := *highest*(*readlist*);
        readlist := $[\perp]^N$;
        forall $q \in \Pi$ **do**
            trigger $\langle$ *ooar.q.self*, *Write* | (*maxts*, *readval*) $\rangle$;

PROOF

Correctness:

Termination – from the termination of the (1,1) atomic register

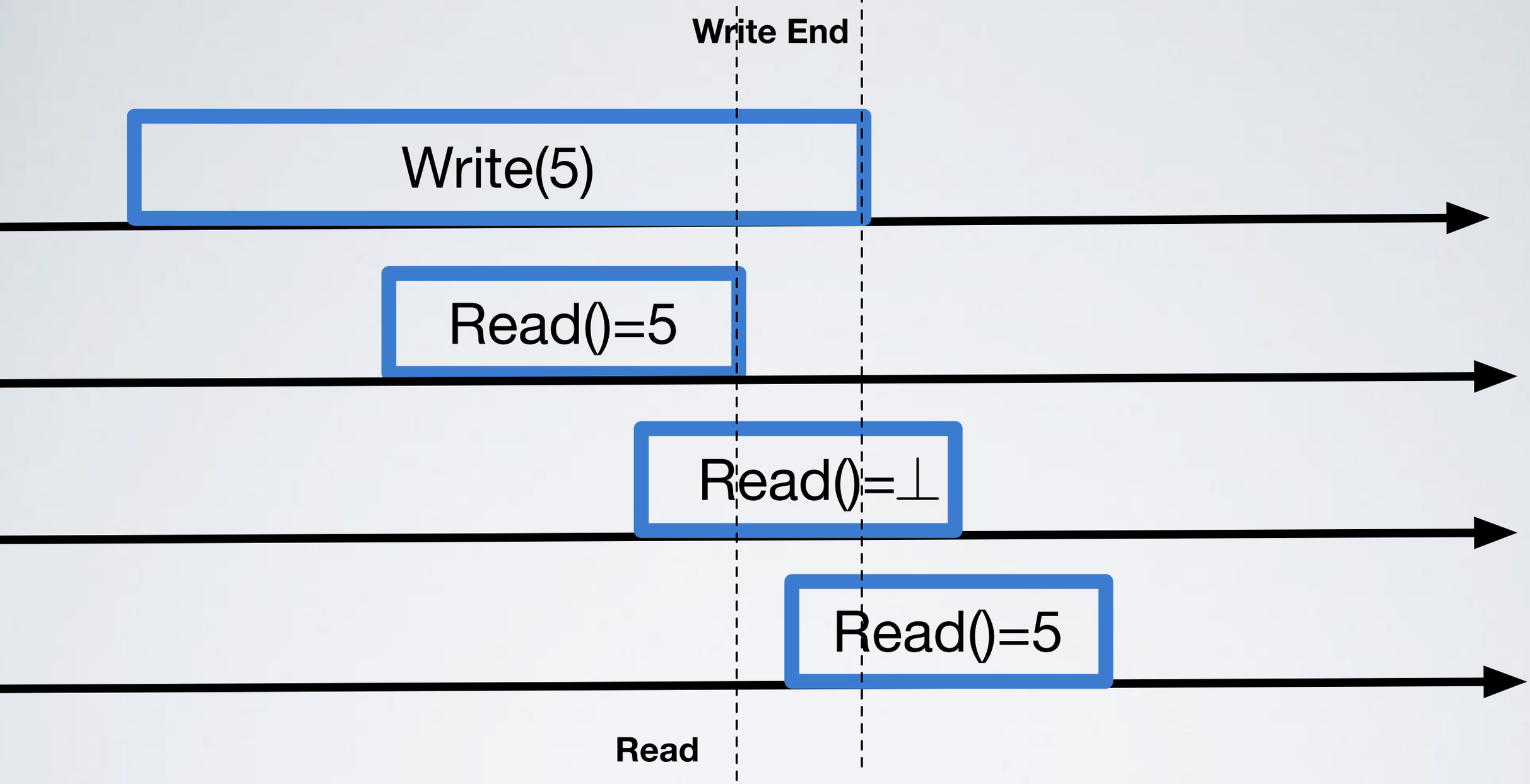
Validity – from the validity of the (1,1) atomic register.

Ordering - Consider a write operation w_1 which writes value v_1 with timestamp s_1. Let w_2 be a write that starts after w_1. Let v_2 and s_2 ($s_1 < s_2$) be the value and the timestamp corresponding to w_2.

Let assume that a read returns v_2: by the algorithm, for each j in [1;N], p_i has written (s_2,v_2) in readers[r; i; j].

For the ordering property of the underlying (1,1) atomic registers, each successive read will return a value with timestamp greater or equal to s2. Then s1 cannot be returned.

LIN. POINTS OF THE ALGORITHM

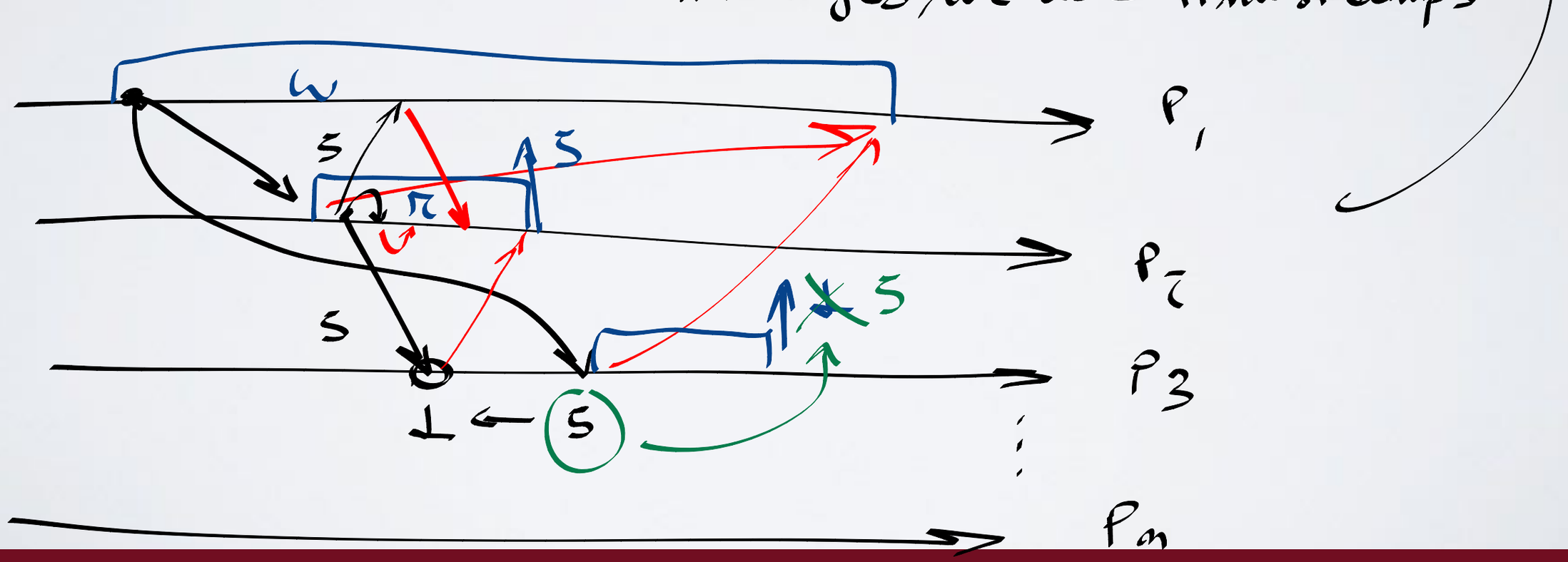


The LP of the write is the minimum between the end of the write and the end of the first read that sees the value of the write.

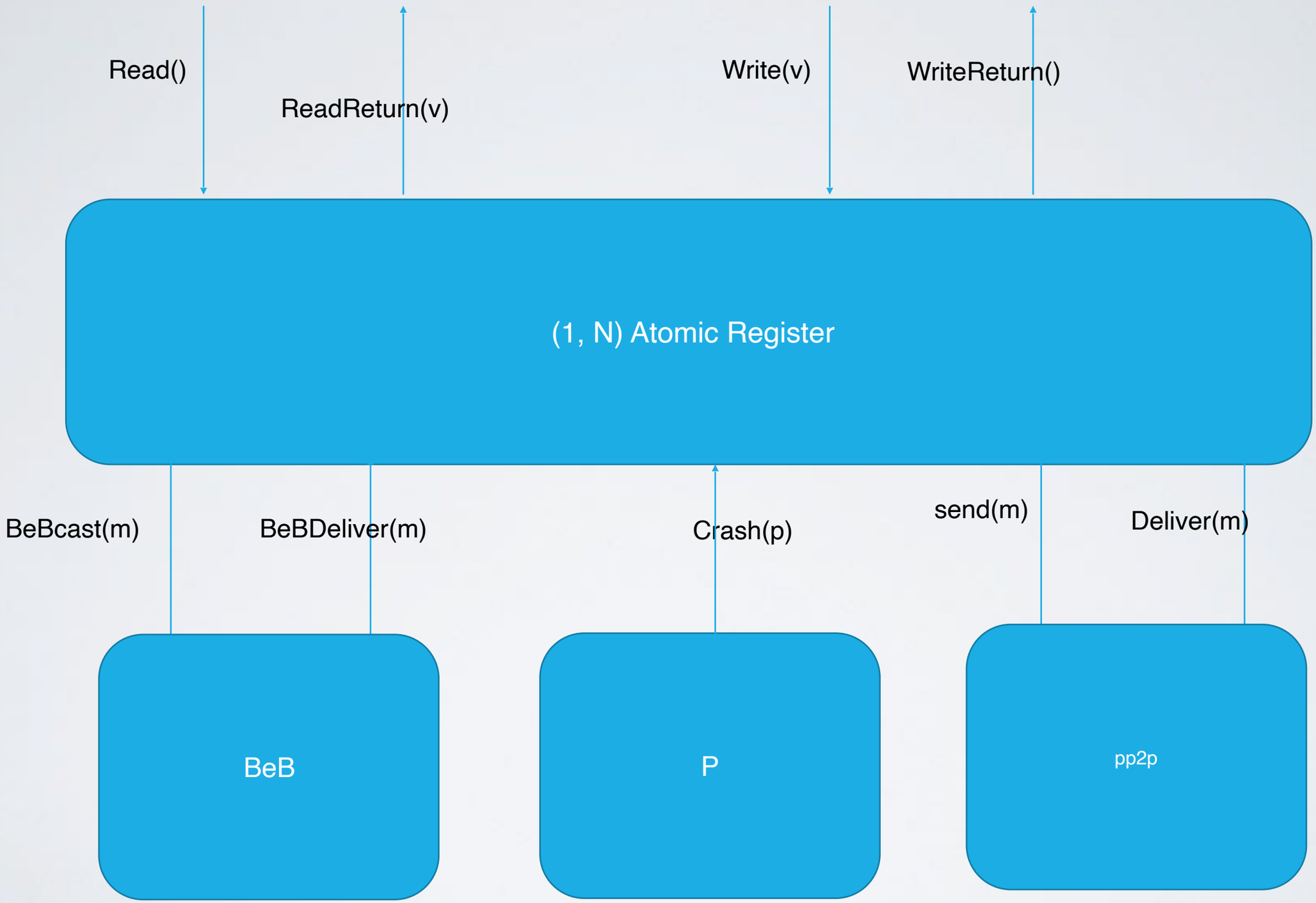
ATOMIC REGISTERS IMPLEMENTATIONS USING MESSAGE PASSING.

can happen that the broadcasting message arrives after p_3(read) but in that case p_2(read) can't terminate because doesn't have all ACKs ⇒ p_2(read) and p_3(read) are concurrent

to ensure that a read with an older value disseminate faulty messages, we use timestamps



READ-IMPOSE WRITE-ALL



READ-IMPOSE WRITE-ALL

The algorithm is a modified version of the Read-One Write-All (1,N) Regular Register

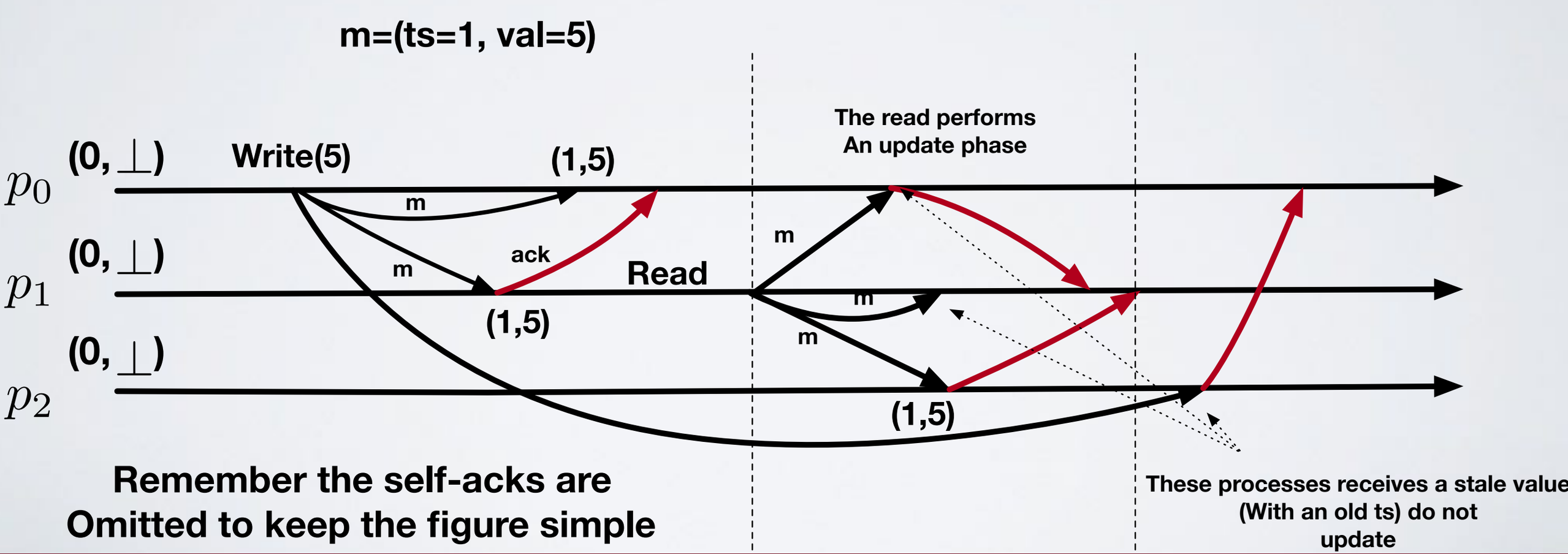
IDEA: “the read operation writes”

The algorithm is called “Read-Impose Write-All” because a read operation imposes to all correct processes to update their local copy of the register with the value read, unless they store a more recent value

READ-IMPOSE WRITE-ALL

Algorithm structure:

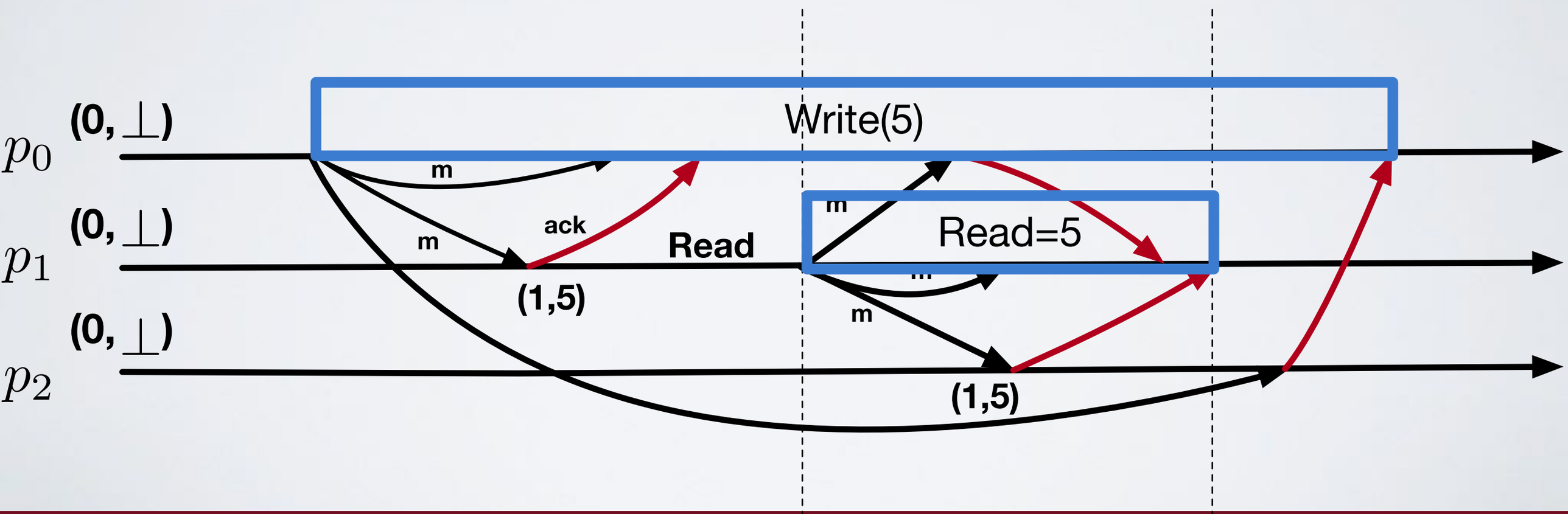
- Write: increase wts, and disseminate a write order to any one
- Read: read locally a (wts,val) , and disseminate a write order to any one
- Process: accept a write if your ts is less than the one in the order. Send ack in any case



READ-IMPOSE WRITE-ALL

Algorithm structure:

- Write: increase wts, and disseminate a write order to any one
- Read: read locally a (wts,val) , and disseminate a write order to any one
- Process: accept a write if your ts is less than the one in the order. Send ack in any case



READ-IMPOSE WRITE-ALL

2M ops WRITE
2M ops READ

upon event $\langle$ *onar*, *Init* $\rangle$ **do**
  $(ts, val) := (0, \perp)$;
  correct := Π;
  writeset := $\emptyset$;
  readval := $\perp$;
  reading := FALSE;

upon event $\langle$ $\mathcal{P}$, *Crash* | p $\rangle$ **do**
  correct := *correct* $\setminus$ $\{p\}$;

upon event $\langle$ *onar*, *Read* $\rangle$ **do**
  reading := TRUE;
  readval := *val*;
  trigger $\langle$ *beb*, *Broadcast* | [WRITE, *ts*, *val*] $\rangle$;

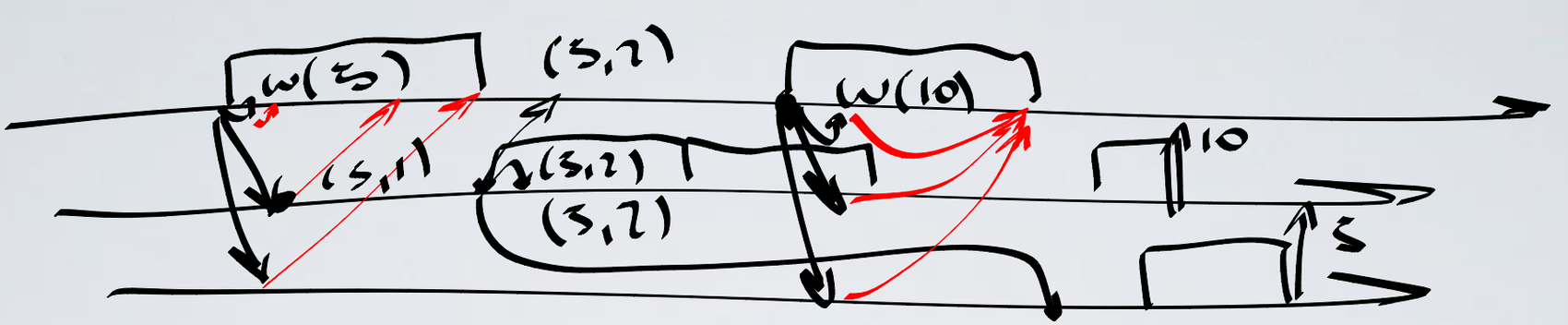
upon event $\langle$ *onar*, *Write* | v $\rangle$ **do**
  trigger $\langle$ *beb*, *Broadcast* | [WRITE, $ts + 1$, v] $\rangle$;

if is sended an older value the message is discarded

upon event $\langle$ *beb*, *Deliver* | p, [WRITE, ts', v'] $\rangle$ **do**
  if $ts' > ts$ **then**
    $(ts, val) := (ts', v')$;
  trigger $\langle$ *pl*, *Send* | p, [ACK] $\rangle$;

upon event $\langle$ *pl*, *Deliver* | p, [ACK] $\rangle$ **then**
  writeset := *writeset* $\cup$ $\{p\}$;

upon *correct* $\subseteq$ *writeset* **do**
  writeset := $\emptyset$;
  if *reading* = TRUE **then**
    reading := FALSE;
    trigger $\langle$ *onar*, *ReadReturn* | *readval* $\rangle$;
  else
    trigger $\langle$ *onar*, *WriteReturn* $\rangle$;



Correctness:

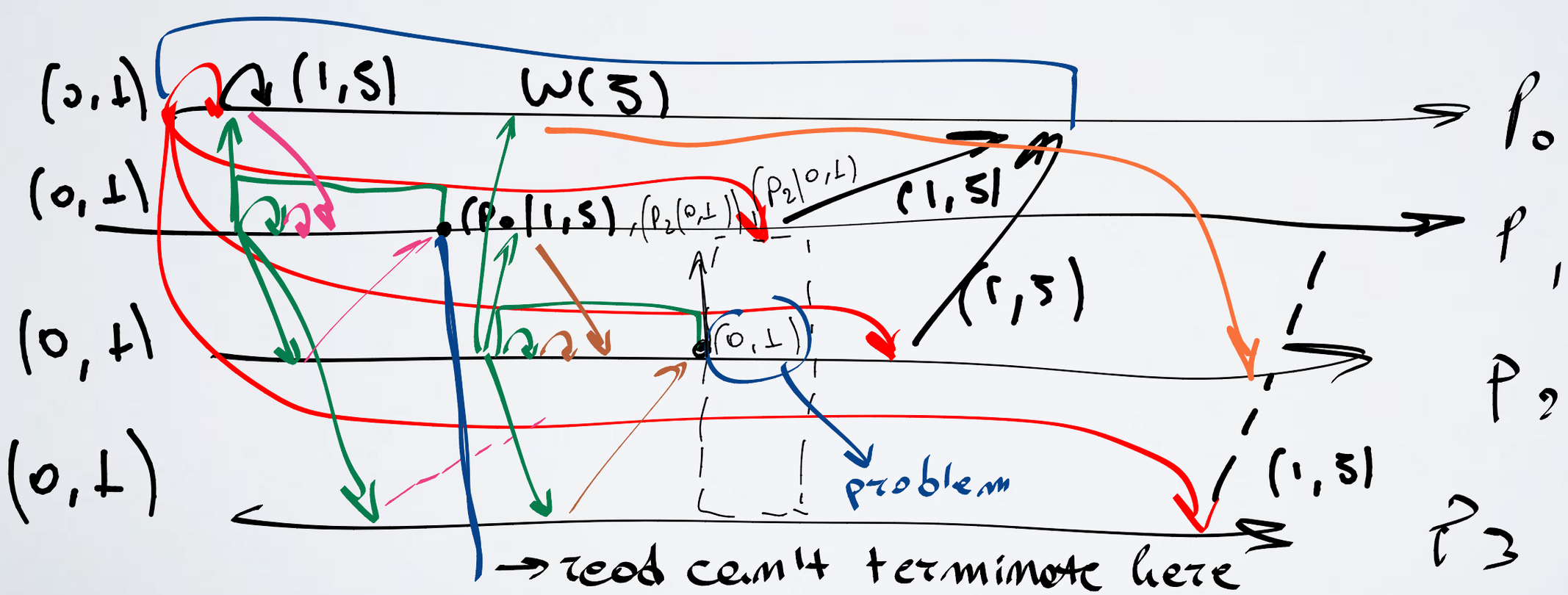
- Termination – as for the Read-One Write-All (1,N) Regular Register.
- Validity - as for Read-One Write-All (1,N) Regular Register.
- Ordering – to complete a read operation, the reader process has to be sure that every other process has in its local copy of the register a value with timestamp bigger or equal of the timestamp of the value read. In this way, any successive read could not return an older value.

Performance:

- Write - a write requests at most 2N messages
- Read - a read requests at most 2N messages
- Number of steps? 2 for write or read

1, N – Atomic Register

IMPLEMENTATION FOR FAIL-SILENT

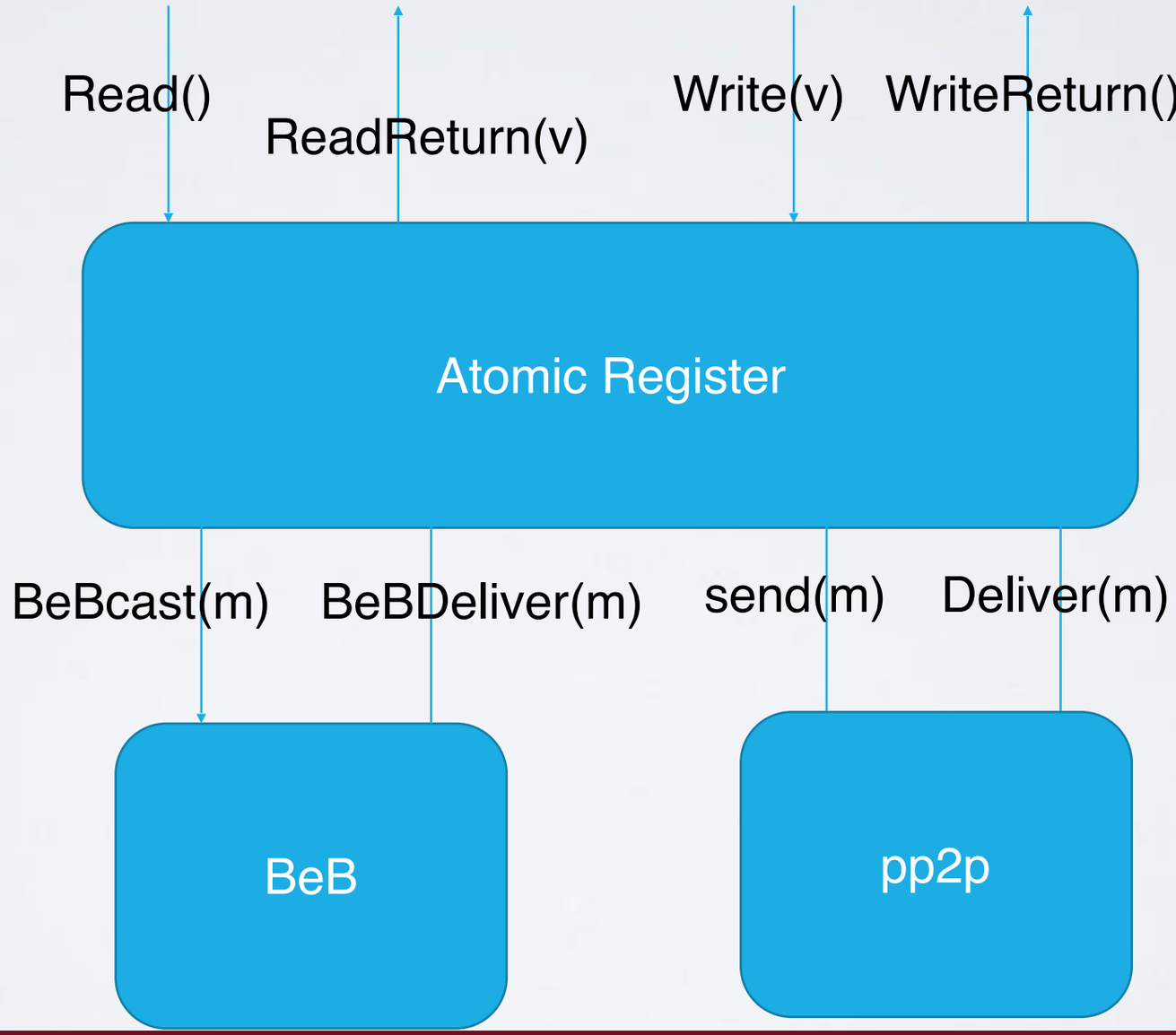


=> impose an additional phase using a broadcast to impose

READ-IMPOSE WRITE-MAJORITY

A majority of correct processes is assumed.

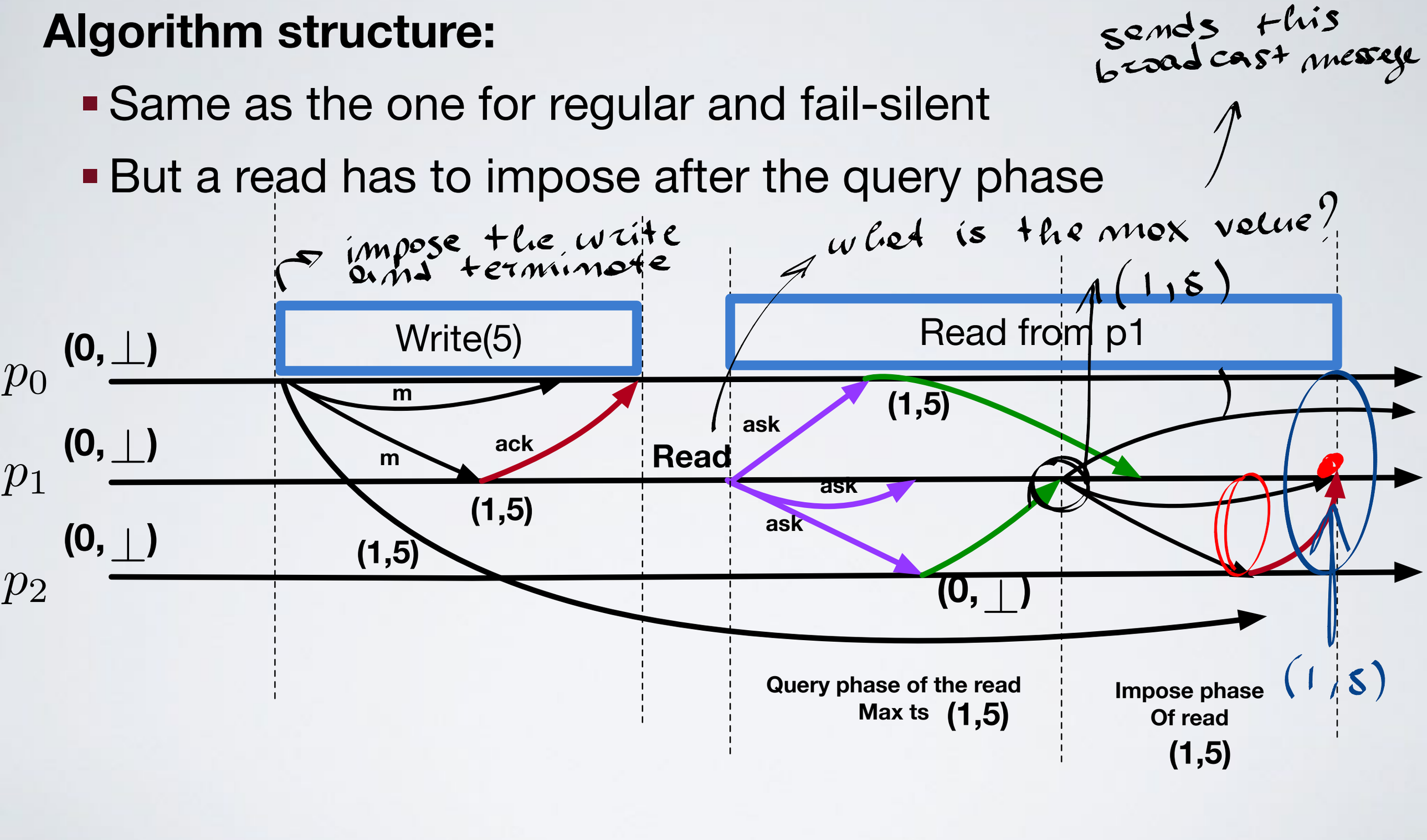
The algorithm is a variation of the Majority Voting (1,N) Regular Register



READ-IMPOSE WRITE-MAJORITY

Algorithm structure:

- Same as the one for regular and fail-silent
- But a read has to impose after the query phase



upon event $\langle$ *onar*, *Init* $\rangle$ **do**
$(ts, val) := (0, \perp)$;
$wts := 0$;
$acks := 0$; ⇝ to end has to be the majority
$rid := 0$;
$readlist := [\perp]^N$;
$readval := \perp$;
$reading :=$ FALSE;

upon event $\langle$ *onar*, *Read* $\rangle$ **do**
$rid := rid + 1$; ⇝ I have to distinguish the acks
$acks := 0$;
$readlist := [\perp]^N$;
$reading :=$ TRUE;
trigger $\langle$ *beb*, *Broadcast* | [READ, *rid*] $\rangle$;

upon event $\langle$ *beb*, *Deliver* | p, [READ, r] $\rangle$ **do**
trigger $\langle$ *pl*, *Send* | p, [VALUE, r, ts, val] $\rangle$;

collecting readings

upon event $\langle$ *pl*, *Deliver* | q, [VALUE, r, ts', v'] $\rangle$ **such that** $r = rid$ **do**
$readlist[q] := (ts', v')$;
if $\#(readlist) > N/2$ **then** → majority
$(maxts, readval) := highest(readlist)$; → take the value with the highest timestamp
$readlist := [\perp]^N$;
trigger $\langle$ *beb*, *Broadcast* | [WRITE, *rid*, *maxts*, *readval*] $\rangle$;

upon event $\langle$ *onar*, *Write* | v $\rangle$ **do**
$rid := rid + 1$;
$wts := wts + 1$;
$acks := 0$;
trigger $\langle$ *beb*, *Broadcast* | [WRITE, rid, wts, v] $\rangle$;

upon event $\langle$ *beb*, *Deliver* | p, [WRITE, r, ts', v'] $\rangle$ **do**
if $ts' > ts$ **then**
$(ts, val) := (ts', v')$;
trigger $\langle$ *pl*, *Send* | p, [ACK, r] $\rangle$;

upon event $\langle$ *pl*, *Deliver* | q, [ACK, r] $\rangle$ **such that** $r = rid$ **do**
$acks := acks + 1$;
if $acks > N/2$ **then**
$acks := 0$;
if *reading* $=$ TRUE **then**
reading := FALSE;
trigger $\langle$ *onar*, *ReadReturn* | *readval* $\rangle$;
else
trigger $\langle$ *onar*, *WriteReturn* $\rangle$;

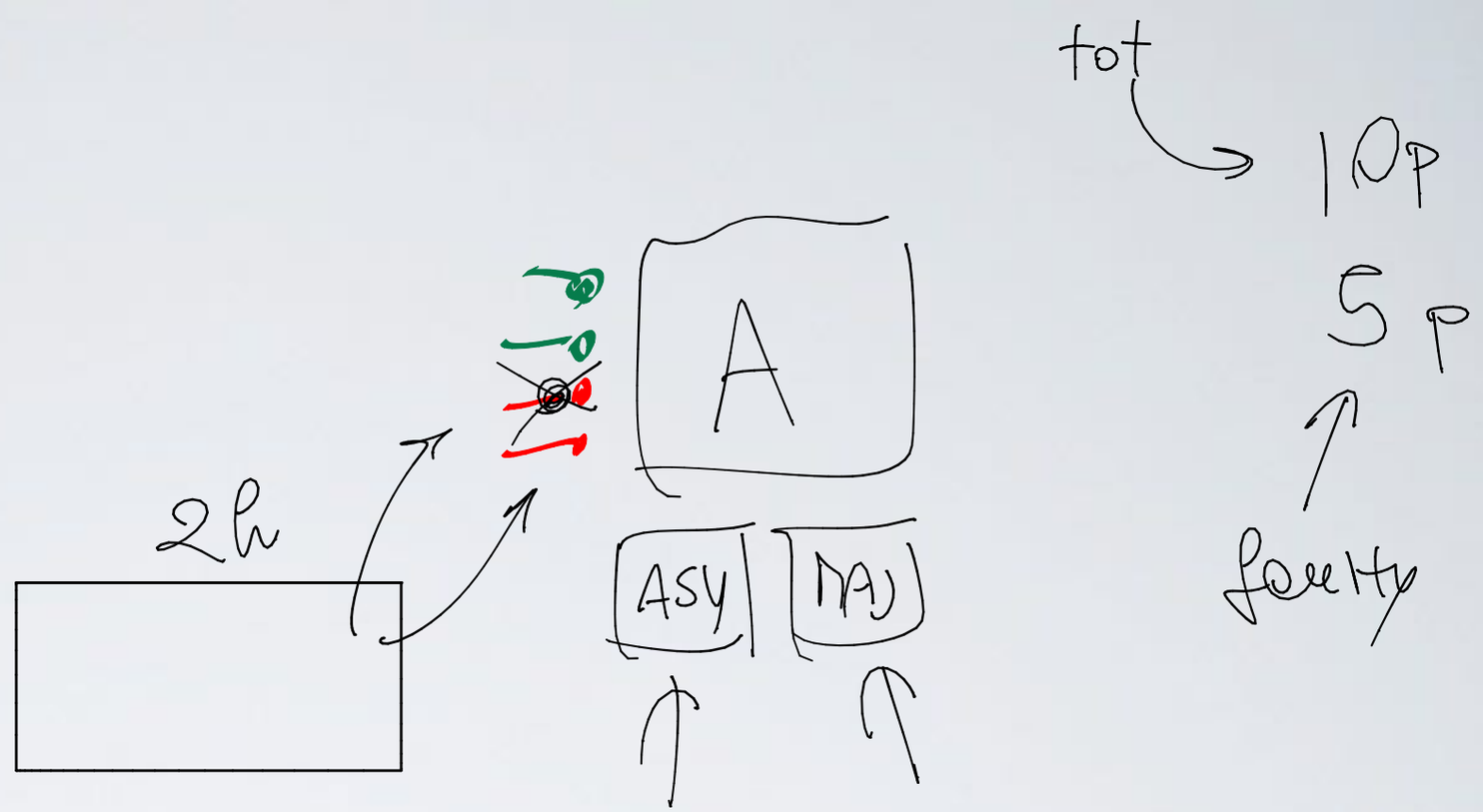
READ-IMPOSE WRITE-MAJORITY

Correctness:

- Termination – as Majority Voting (1,N) Regular Register
- Validity – as Majority Voting (1,N) Regular Register.
- Ordering – due to the fact that the read imposes the write of the value read to a majority of processes and to the property of intersection of quorums.

Performance:

- Write – at most 2N messages
- Read – at most 4N messages
- Step complexity: Write 2 steps, Read 4 steps.



COMPARISONS BETWEEN ALGORITHMS

REGULAR VS ATOMIC: PERFORMANCES

MESSAGE PASSING IMPLEMENTATIONS!

number of failures that the system can tollerate

		(1,n)-Regular Register		(1,n)-Atomic Register	
		READ	WRITE	READ	WRITE
Fail-Stop (P)	Messages	0 (local)	2N	2N	2N
	Steps	0 (local)	2	2	2
	Resiliency	f<N		f<N	
Fail-Silent	Messages	2N	2N	4N	2N
	Steps	2	2	4	2
	Resiliency	f<N/2		f<N/2	

(N,N)- ATOMIC REGISTER

(N,N)-ATOMIC REGISTER SPEC

Module 4.3: Interface and properties of an (N, N) atomic register

Module:

Name: (N, N)-AtomicRegister, **instance** *nnar*.

Events:

Request: $\langle$ *nnar*, *Read* $\rangle$: Invokes a read operation on the register.

Request: $\langle$ *nnar*, *Write* | v $\rangle$: Invokes a write operation with value v on the register.

Indication: $\langle$ *nnar*, *ReadReturn* | v $\rangle$: Completes a read operation on the register with return value v.

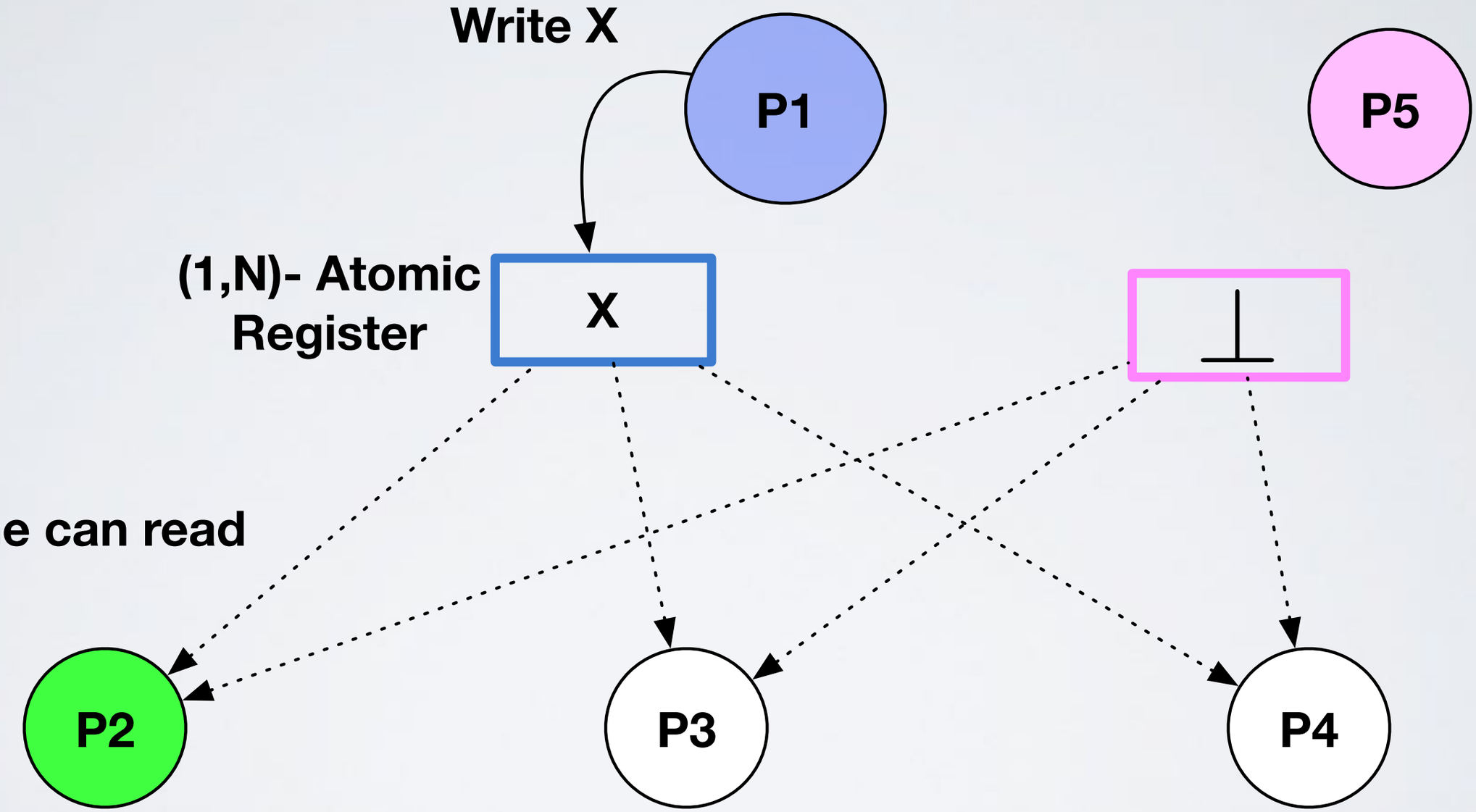
Indication: $\langle$ *nnar*, *WriteReturn* $\rangle$: Completes a write operation on the register.

Properties:

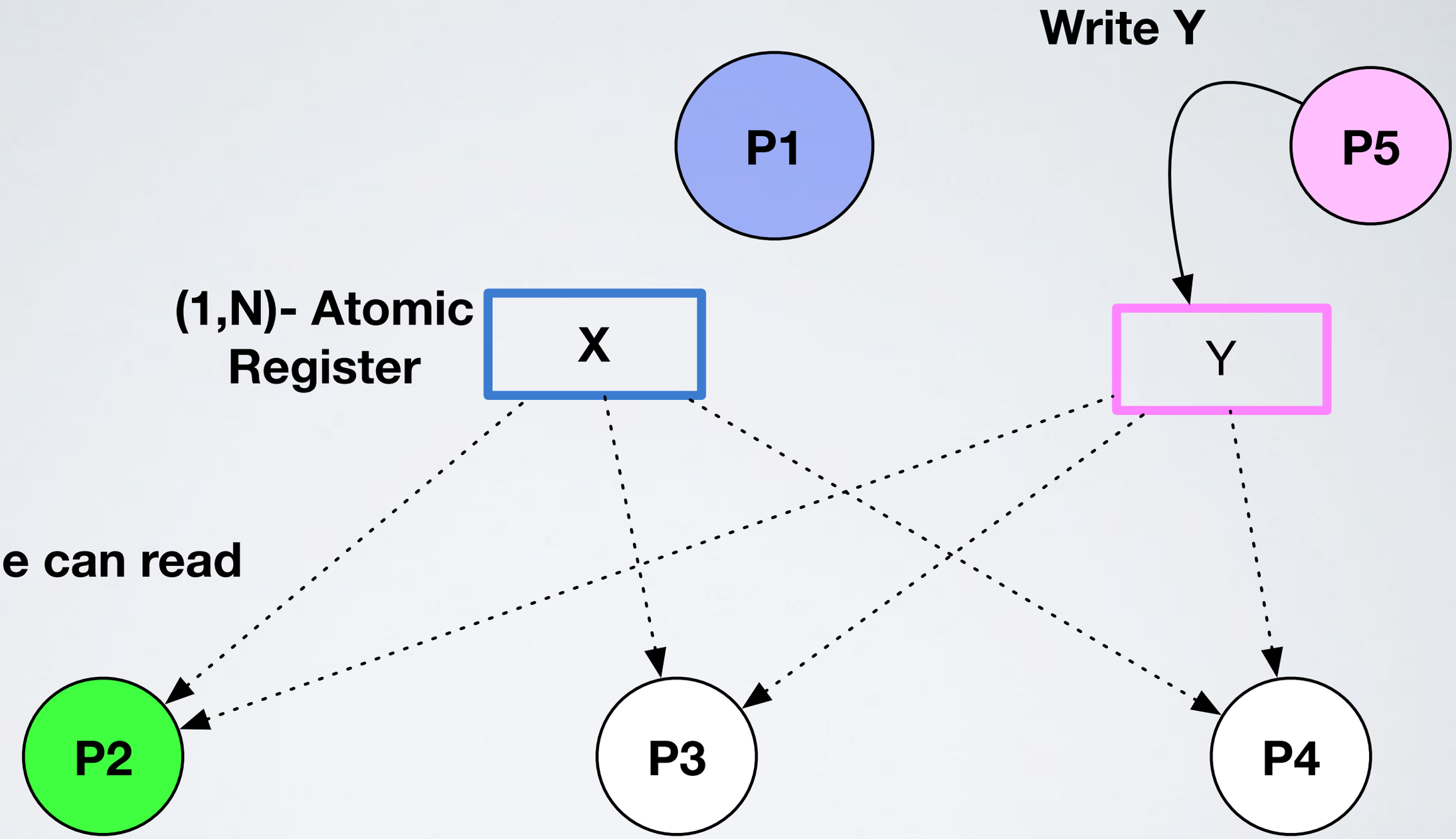
NNAR1: *Termination:* Same as property ONAR1 of a $(1, N)$ atomic register (Module 4.2).

NNAR2: *Atomicity:* Every read operation returns the value that was written most recently in a hypothetical execution, where every failed operation appears to be complete or does not appear to have been invoked at all, and every complete operation appears to have been executed at some instant between its invocation and its completion.

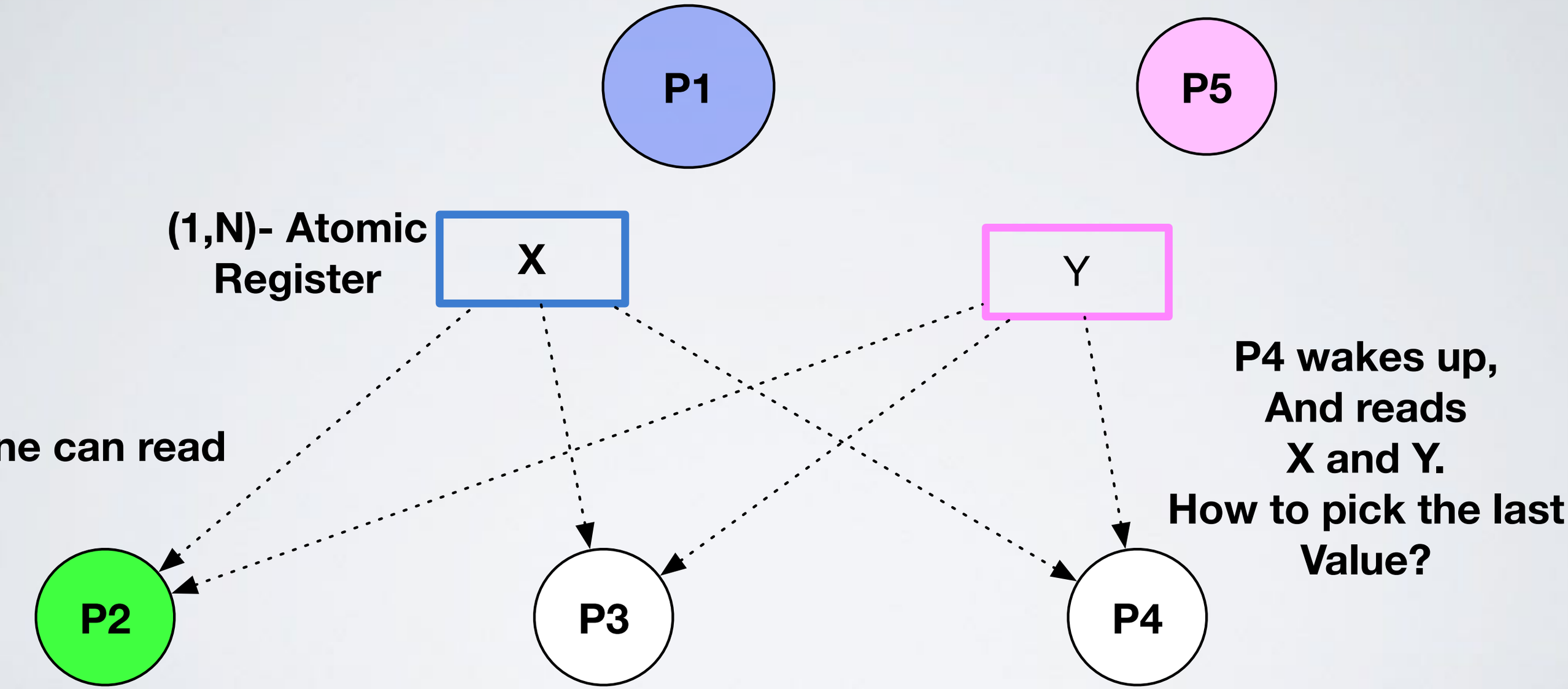
(N,N)-ATOMIC SIMULATION USING (1,N)-ATOMIC



(N,N)-ATOMIC SIMULATION USING (1,N)-ATOMIC



(N,N)-ATOMIC SIMULATION USING (1,N)-ATOMIC

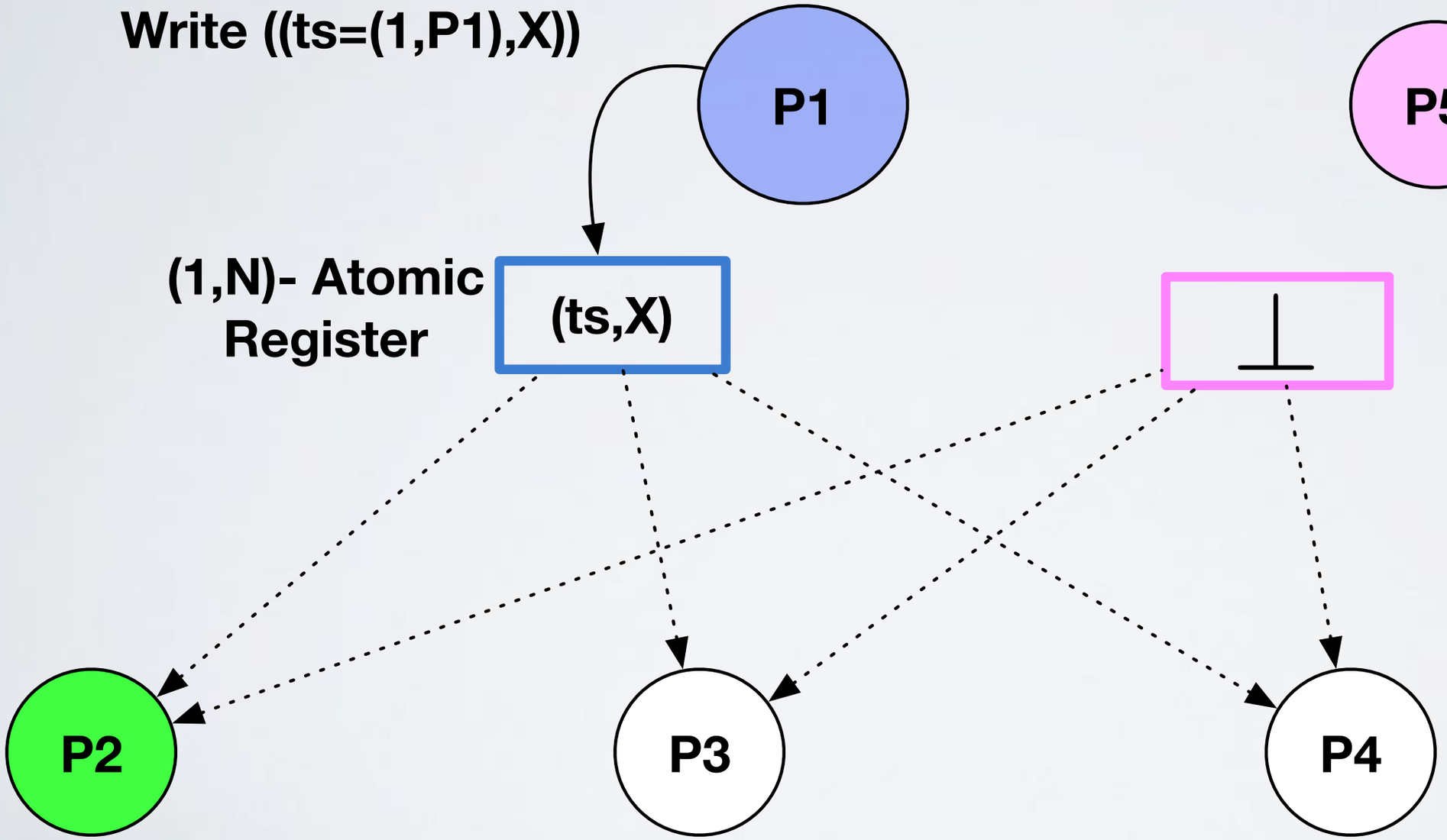


(N,N)-ATOMIC SIMULATION USING (1,N)-ATOMIC

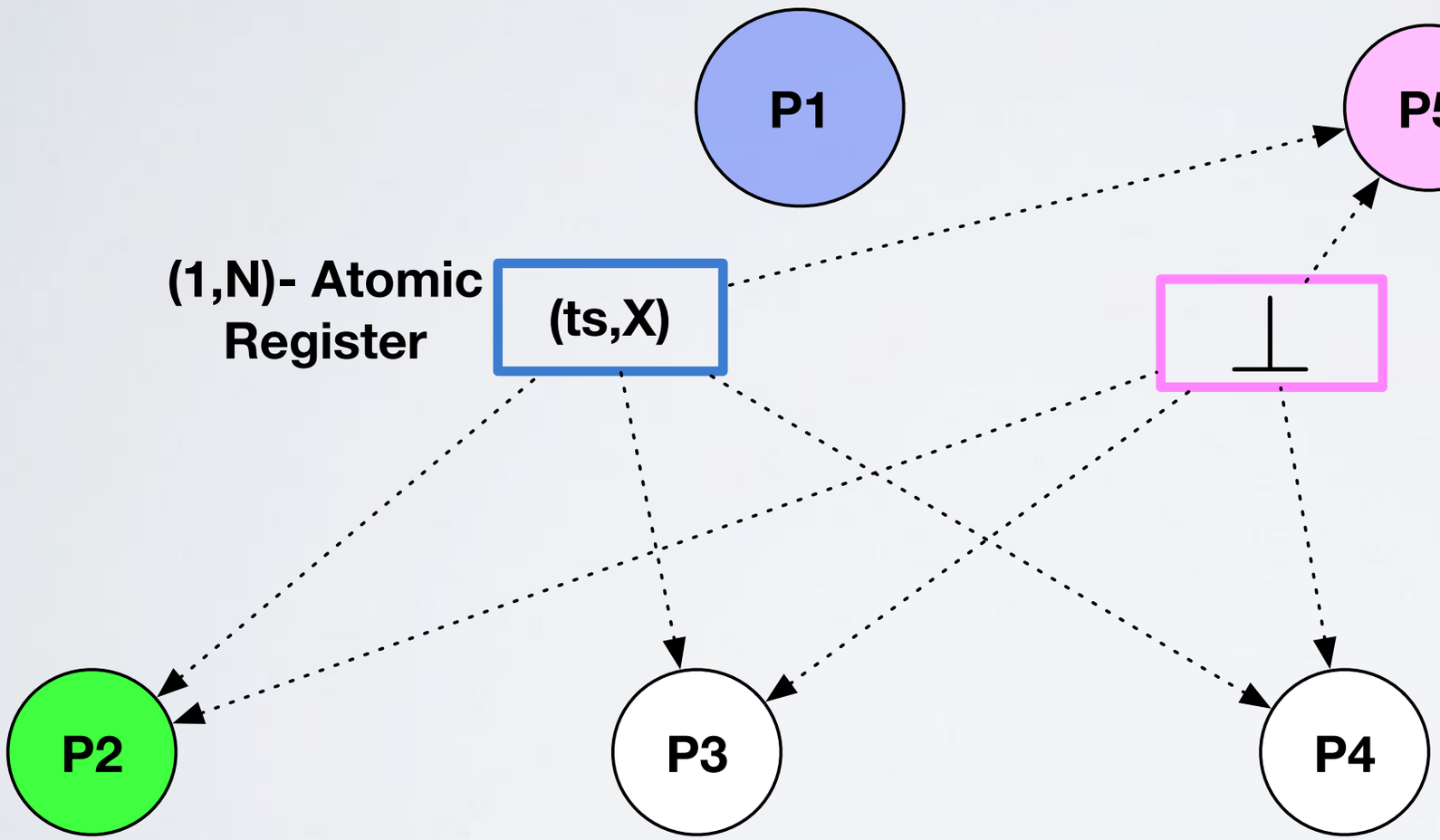
ts= <wtd,id_writer>

Usual trick: we use IDS to break ties, if two TSs have equal wts, pick the smallest ID

Write ((ts=(1,P1),X))



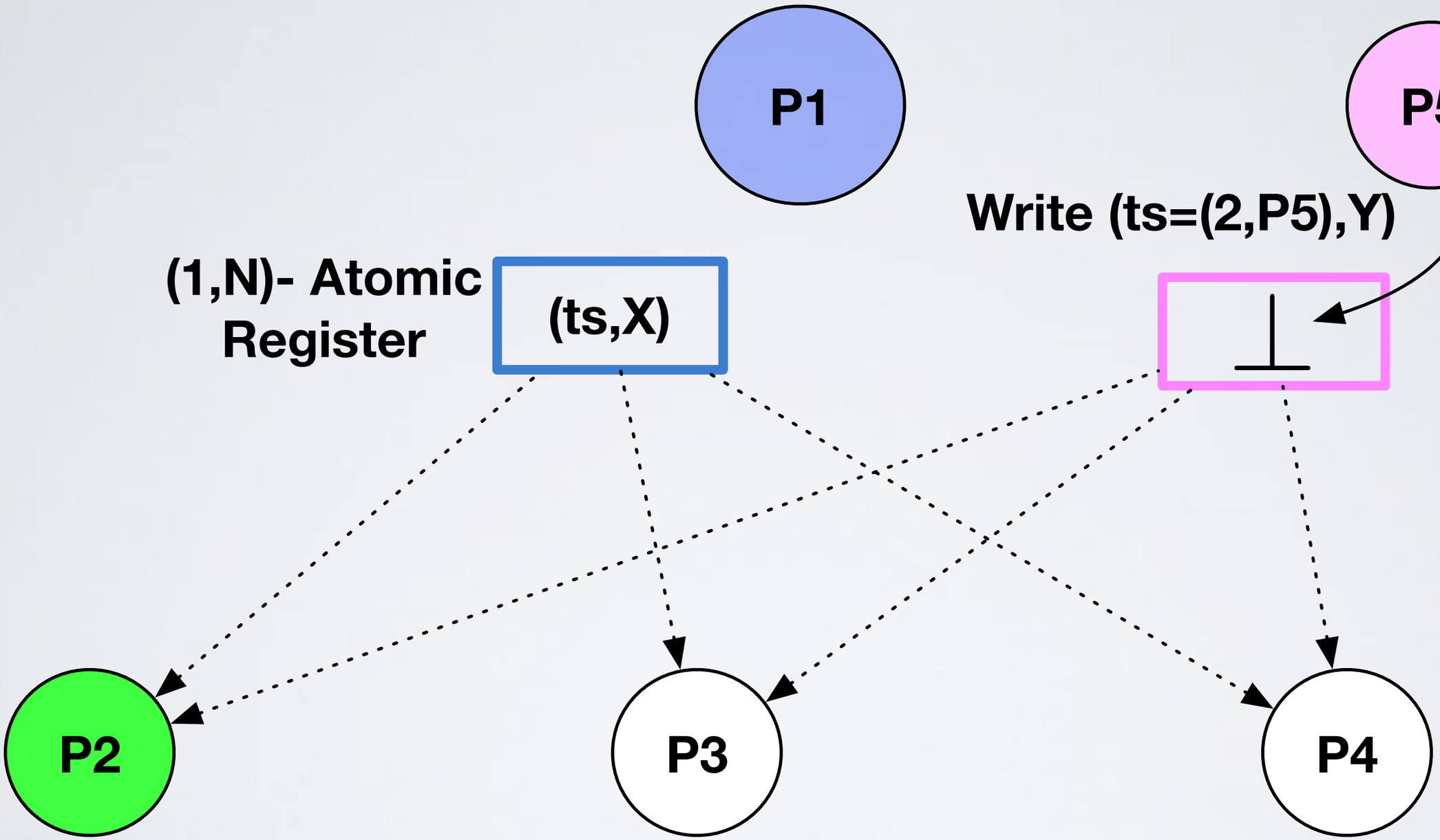
(N,N)-ATOMIC SIMULATION USING (1,N)-ATOMIC



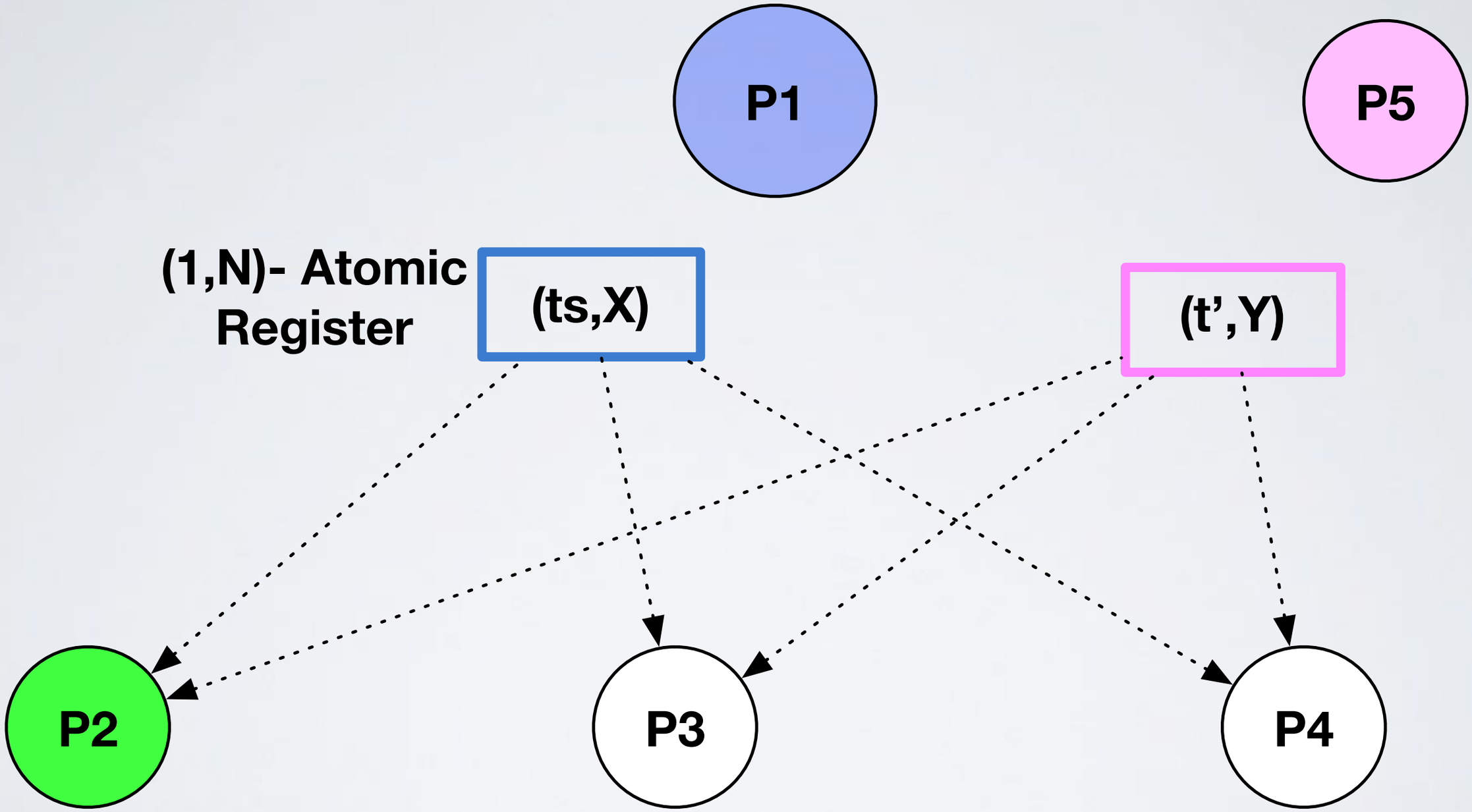
A writer has a consult phase
In which it reads all registers

wts_max=1

(N,N)-ATOMIC SIMULATION USING (1,N)-ATOMIC



(N,N)-ATOMIC SIMULATION USING (1,N)-ATOMIC



(N,N)-ATOMIC SIMULATION USING (1,N)-ATOMIC

upon event $\langle$ *nnar*, *Init* $\rangle$ **do**
    $val := \perp$;
    writing := FALSE;
    readlist := $[\perp]^N$;
    forall $q \in \Pi$ **do**
        Initialize a new instance *onar.q* of $(1, N)$-AtomicRegister
            with writer q;

upon event $\langle$ *nnar*, *Write* | v $\rangle$ **do**
    $val := v$;
    writing := TRUE;
    forall $q \in \Pi$ **do**
        trigger $\langle$ *onar.q*, *Read* $\rangle$;

upon event $\langle$ *nnar*, *Read* $\rangle$ **do**
    forall $q \in \Pi$ **do**
        trigger $\langle$ *onar.q*, *Read* $\rangle$;

upon event $\langle$ *onar.q*, *ReadReturn* | (ts', v') $\rangle$ **do**
    readlist$[q]$:= $(ts', rank(q), v')$;
    if $\#(readlist) = N$ **then**
        (ts, v) := *highest*(*readlist*);
        readlist := $[\perp]^N$;
        if *writing* = TRUE **then**
            writing := FALSE;
            trigger $\langle$ *onar.self*, *Write* | $(ts + 1, val)$ $\rangle$
        else
            trigger $\langle$ *nnar*, *ReadReturn* | v $\rangle$;

upon event $\langle$ *onar.self*, *WriteReturn* $\rangle$ **do**
    trigger $\langle$ *nnar*, *WriteReturn* $\rangle$;

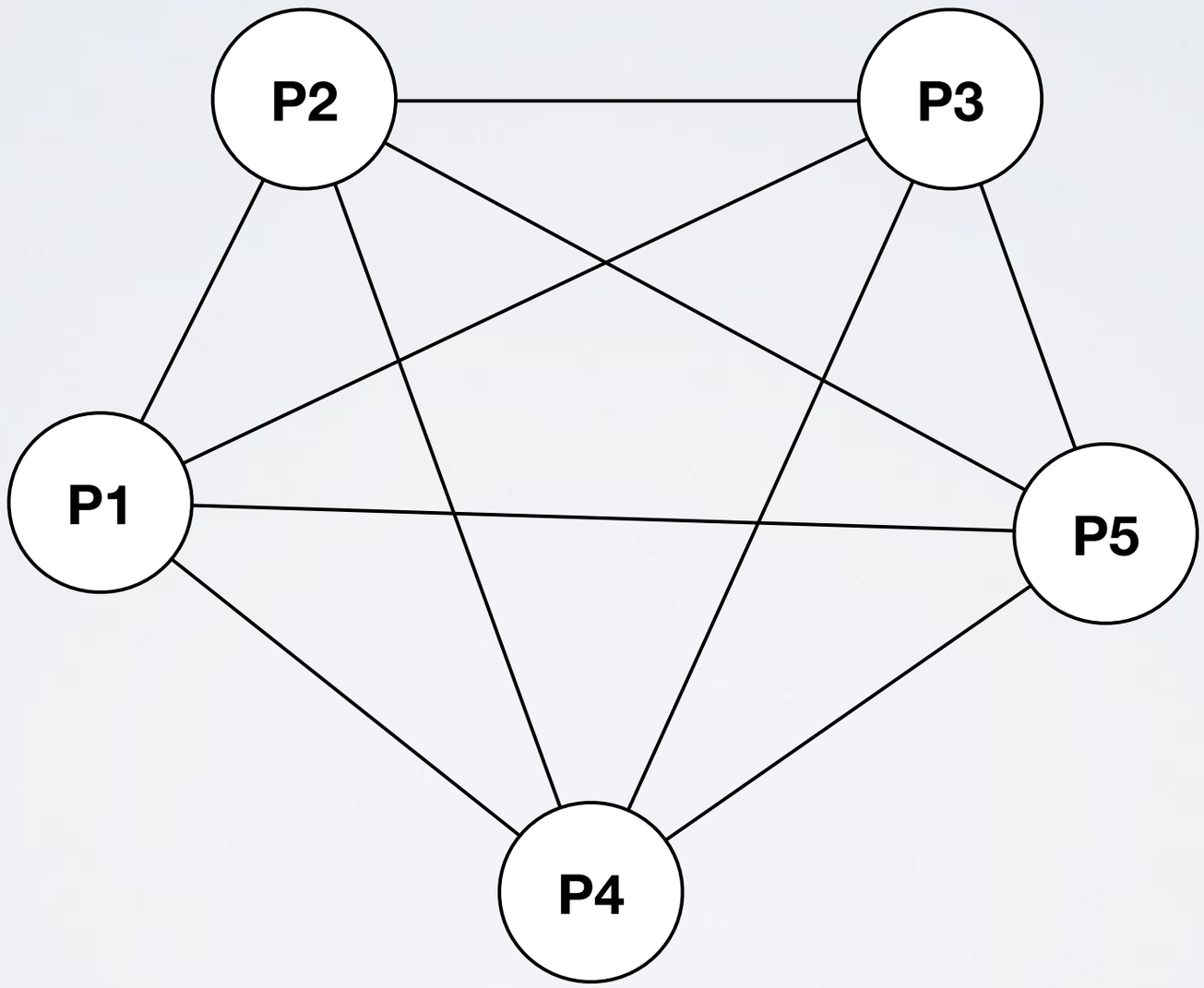
QUESTIONS

Is it possible to implement (N,N)-AR on message passing using P?

Is it possible to implement (N,N)-AR on fail-silent assuming majority of correct processes?

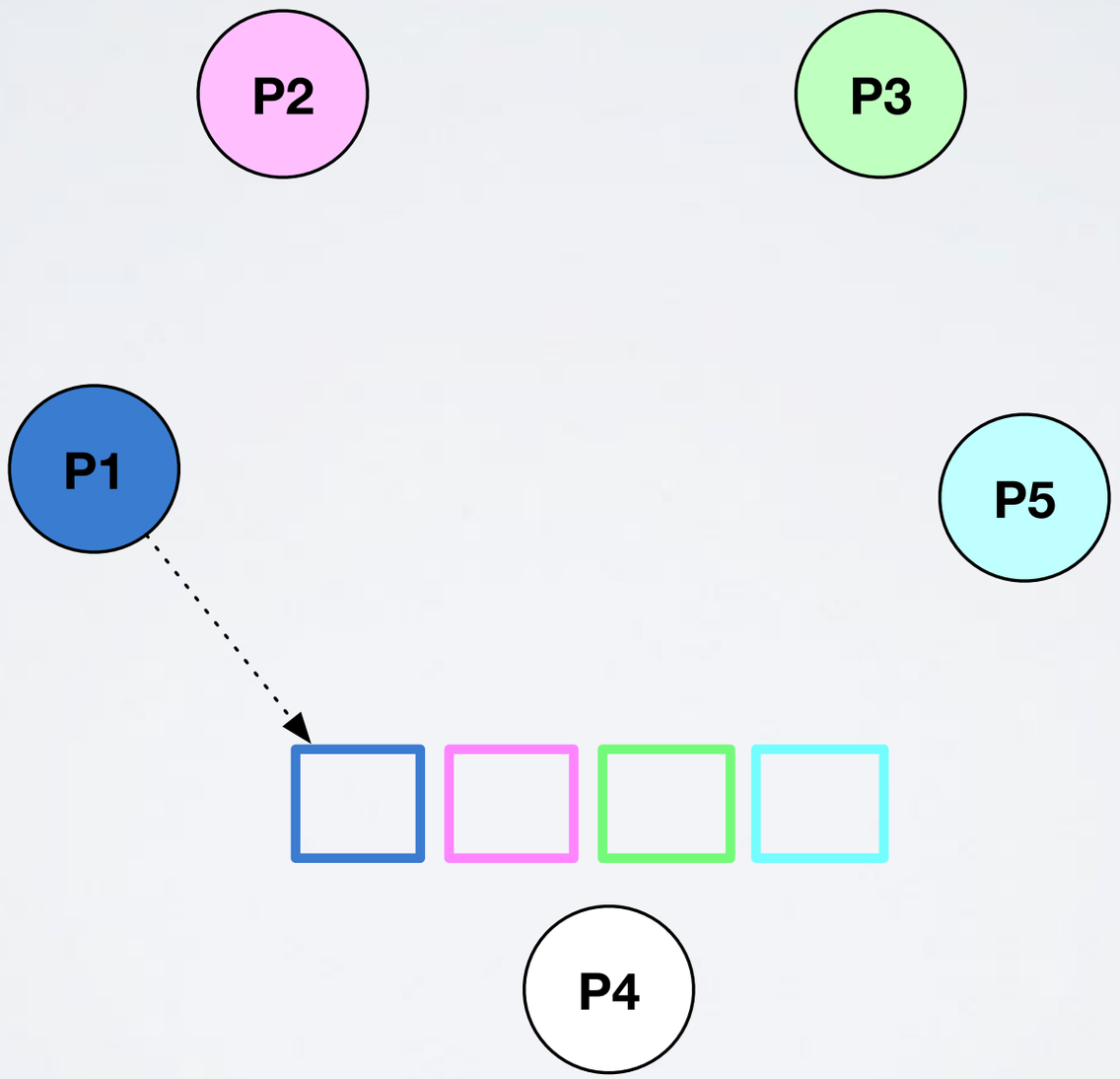
QUESTIONS

Could you show how to simulate message passing using a (1,1)-atomic registers?



QUESTIONS

Could you show how to simulate message passing using a (1,1)-atomic registers?



EXERCISE

Exercise 4.7: *Give an algorithm that implements a* $(1,1)$ *atomic register in the fail-silent model and that is more efficient than the "Read-Impose Write-Majority" algorithm (Algorithm 4.6–4.7, which implements a* $(1,N)$ *atomic register in the fail-silent model).*

EXERCISE

Exercise 4.7: *Give an algorithm that implements a* $(1,1)$ *atomic register in the fail-silent model and that is more efficient than the "Read-Impose Write-Majority" algorithm (Algorithm 4.6–4.7, which implements a* $(1,N)$ *atomic register in the fail-silent model).*

Algorithm 4.20: Modification of Majority Voting to Implement a $(1,1)$ Atomic Register

upon event $\langle$ *onrr*, *Read* $\rangle$ **do**
    $rid := rid + 1$;
    $readlist := [\perp]^N$;
    $readlist[self] := (ts, val)$;
    trigger $\langle$ *beb*, *Broadcast* | [READ, *rid*] $\rangle$;

upon event $\langle$ *beb*, *Deliver* | p, [READ, r] $\rangle$ **do**
    if $p \neq self$ **then**
        trigger $\langle$ *pl*, *Send* | p, [VALUE, r, ts, val] $\rangle$;

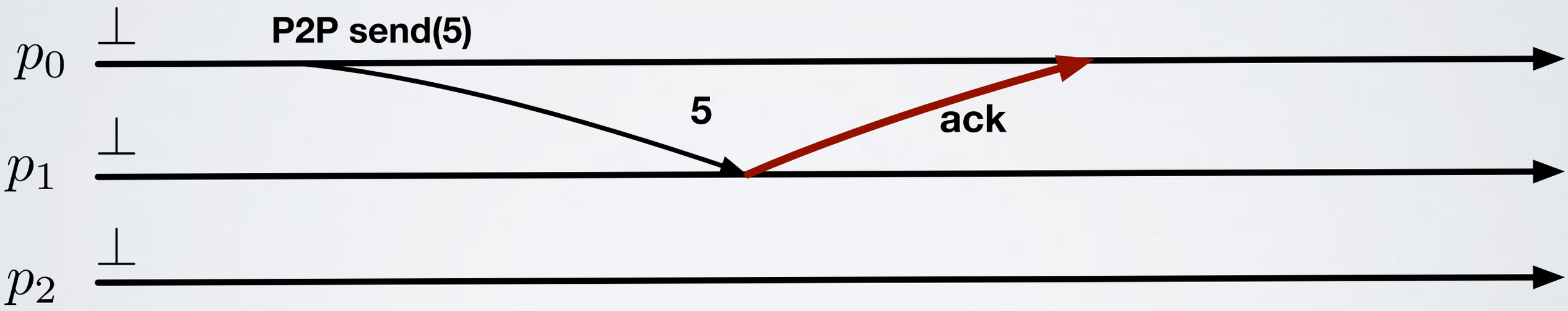
upon event $\langle$ *pl*, *Deliver* | q, [VALUE, r, ts', v'] $\rangle$ **such that** $r = rid$ **do**
    $readlist[q] := (ts', v')$;
    if $\#(readlist) > N/2$ **then**
        $(ts, val) := highest(readlist)$;
        $readlist := \emptyset$;
        trigger $\langle$ *onrr*, *ReadReturn* | val $\rangle$;

EXERCISE

Exercise 4.7: *Give an algorithm that implements a* $(1,1)$ *atomic register in the fail-silent model and that is more efficient than the "Read-Impose Write-Majority" algorithm (Algorithm 4.6–4.7, which implements a* $(1,N)$ *atomic register in the fail-silent model).*

Why not using just this pattern? It Is very efficient
Each write is 2 messages

Each write send to the only reader P1, and wait for ack.



EXERCISE

Exercise 4.6: *Explain why the request identifier included in* WRITE, ACK, READ, *and* VALUE *messages of the "Majority Voting" algorithm (Algorithm 4.2) can be omitted in the "Read-One Write-All" algorithm (Algorithm 4.1).*

Read-one Write-all Algorithm

upon event $\langle$ *onrr*, *Init* $\rangle$ **do**
    $val := \bot$;
    $correct := \Pi$;
    $writeset := \emptyset$;

upon event $\langle\ \mathcal{P}$, *Crash* | $p\ \rangle$ **do**
    $correct := correct \setminus \{p\}$;

upon event $\langle$ *onrr*, *Read* $\rangle$ **do**
    trigger $\langle$ *onrr*, *ReadReturn* | $val\ \rangle$;

upon event $\langle$ *onrr*, *Write* | $v\ \rangle$ **do**
    trigger $\langle$ *beb*, *Broadcast* | [WRITE, v] $\rangle$;

upon event $\langle$ *beb*, *Deliver* | q, [WRITE, v] $\rangle$ **do**
    $val := v$;
    trigger $\langle$ *pl*, *Send* | q, ACK $\rangle$;

upon event $\langle$ *pl*, *Deliver* | p, ACK $\rangle$ **do**
    $writeset := writeset \cup \{p\}$;

upon $correct \subseteq writeset$ **do**
    $writeset := \emptyset$;
    trigger $\langle$ *onrr*, *WriteReturn* $\rangle$;

EXERCISE

Exercise 4.6: *Explain why the request identifier included in* WRITE, ACK, READ, *and* VALUE *messages of the "Majority Voting" algorithm (Algorithm 4.2) can be omitted in the "Read-One Write-All" algorithm (Algorithm 4.1).*

Read-one Write-all Algorithm

upon event $\langle$ *onrr*, *Init* $\rangle$ **do**
  $(ts, val) := (0, \perp)$;
  $wts := 0$;
  acks $:= 0$;
  rid $:= 0$;
  readlist $:= [\perp]^N$;

upon event $\langle$ *onrr*, *Write* | v $\rangle$ **do**
  $wts := wts + 1$;
  acks $:= 0$;
  trigger $\langle$ *beb*, *Broadcast* | [WRITE, wts, v] $\rangle$;

upon event $\langle$ *beb*, *Deliver* | p, [WRITE, ts', v'] $\rangle$ **do**
  if $ts' > ts$ **then**
    $(ts, val) := (ts', v')$;
  trigger $\langle$ *pl*, *Send* | p, [ACK, ts'] $\rangle$;

upon event $\langle$ *pl*, *Deliver* | q, [ACK, ts'] $\rangle$ **such that** $ts' = wts$ **do**
  acks $:=$ *acks* $+ 1$;
  if *acks* $> N/2$ **then**
    acks $:= 0$;
    trigger $\langle$ *onrr*, *WriteReturn* $\rangle$;

upon event $\langle$ *onrr*, *Read* $\rangle$ **do**
  rid $:=$ *rid* $+ 1$;
  readlist $:= [\perp]^N$;
  trigger $\langle$ *beb*, *Broadcast* | [READ, *rid*] $\rangle$;

upon event $\langle$ *beb*, *Deliver* | p, [READ, r] $\rangle$ **do**
  trigger $\langle$ *pl*, *Send* | p, [VALUE, r, ts, val] $\rangle$;

upon event $\langle$ *pl*, *Deliver* | q, [VALUE, r, ts', v'] $\rangle$ **such that** $r =$ *rid* **do**
  readlist$[q] := (ts', v')$;
  if $\#$(*readlist*) $> N/2$ **then**
    $v :=$ *highestval*(*readlist*);
    readlist $:= [\perp]^N$;
    trigger $\langle$ *onrr*, *ReadReturn* | v $\rangle$;