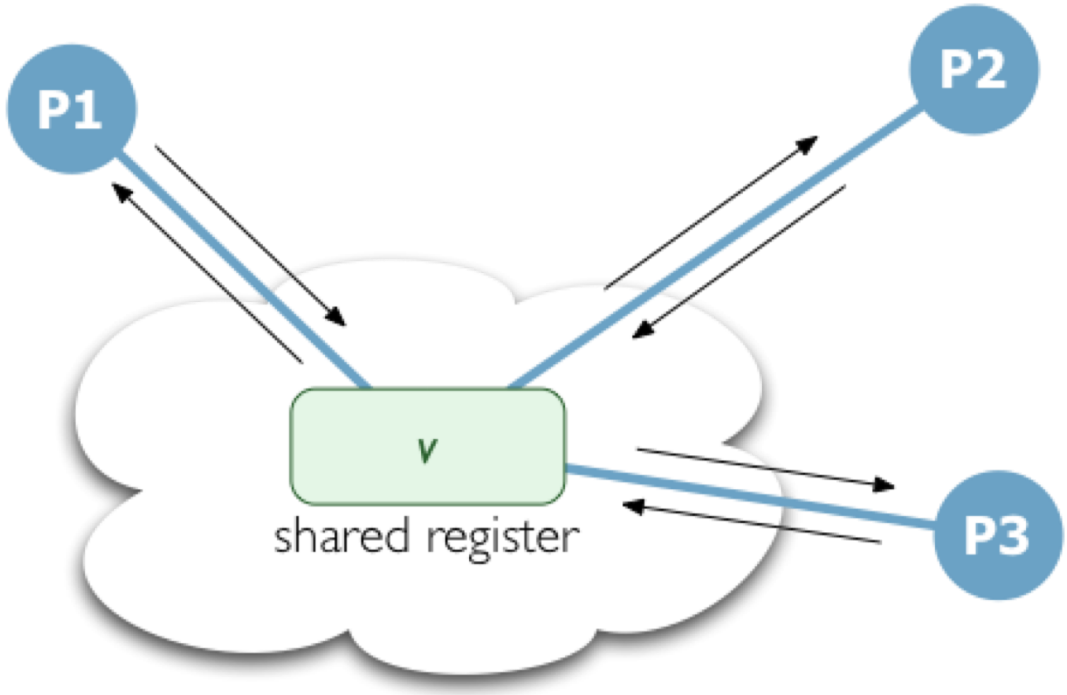
Distributed Systems
Master of Science in Engineering in Computer Science

AA 2018/2019

LECTURES 12-13: REGISTERS

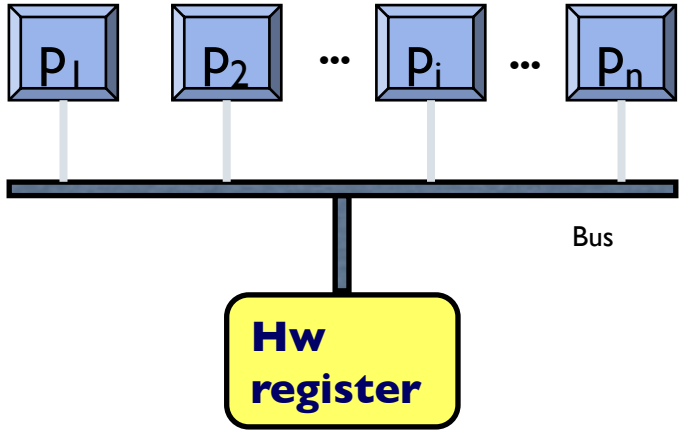
Register: definition

A *register* is a shared variable accessed by processes through *read* and *write* operations



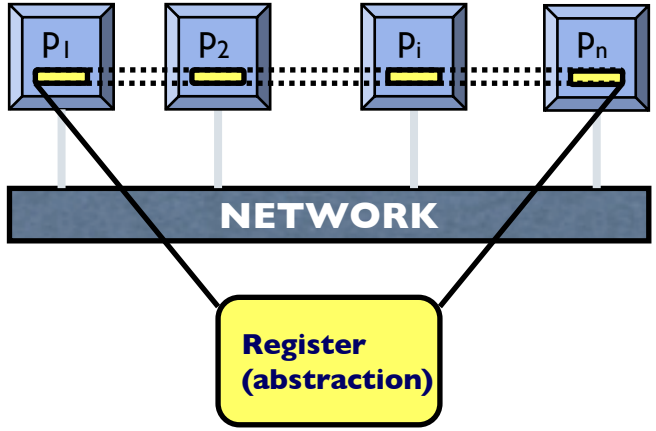
Distributed Systems: register abstraction

- **Multiprocessor machine**:
  - Processes typically communicate through registers at hardware level



- The set of these registers constitute the physical memory

- **Distributed message passing system:**
  - no physical shared memory
  - Processes communicate exchanging msg over a network



- **Register abstraction support the design of distributed solution, by hiding the complexity of the underlying message passing system and the distribution of the data**

Register operations

A process accesses a register through:

- **Read operation**, `read()`$\rightarrow$`v`: it returns the "current" value v of the register; this operation does not modify the content of the register;
- **Write operation**, `write(v)` : it writes the value `v` in the register and returns `true` at the end of the operation

Each operation starts with an invocation and terminates when the corresponding response is received

Register: Assumption

- A register stores only positive integers and it is initialize to 0
- Each value written is univocally identified
- Processes are sequential: a process cannot invoke a new operation before the one it previously invoked (if any) returned

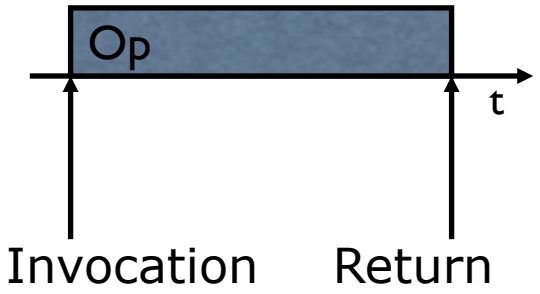
Register: Notation

(X,Y) denotes a register where X processes can write and Y processes can read

- (1,1) denotes a register where only a process can write and only a process can read. It is a priori known which process can write and which can read
- (1,N) denotes a register where a single process, a priori known, can write, and N processes can read

Operations

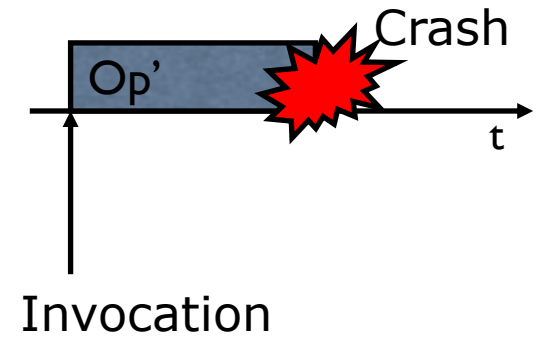
- Every operation is characterized by two events:
  - Invocation
  - Return (Confirmation for the write operation and a value for the read)



- Each of these events occur at a single indivisible point of time

- An operation is *complete* if both the invocation and the return events are occurred

- A *Failed* operation is an operation invoked by some process p_i that crashes before obtaining a return



Precedence between Operations

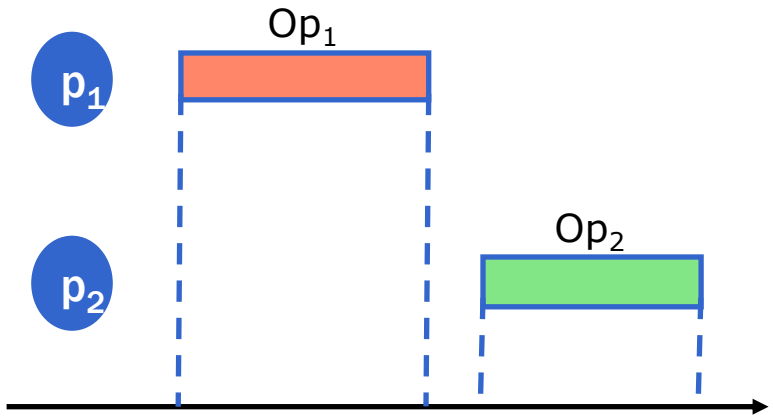
- The execution of an operation invoked by a process p, is the time interval defined by the invocation event and the return event

- Given two operations o e o', o precedes o' if the response event of o precedes the invocation event of o'

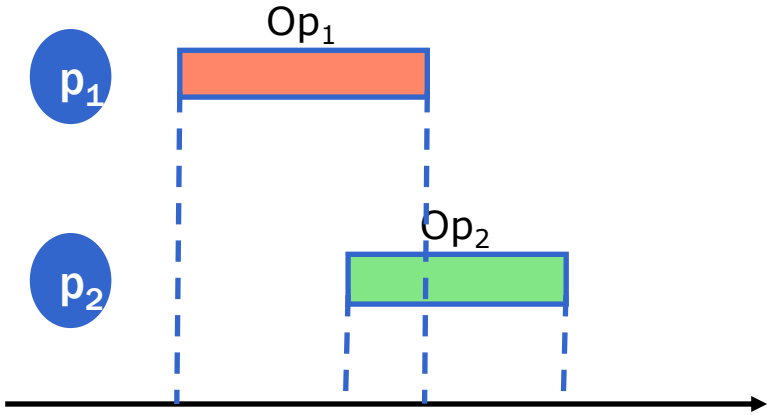
- An operation o invoked by a process p may precedes an operation o' invoked by p' only if o completes

- If it is not possible to define a precedence relation between two operations, they are said to be concurrent

Example



Op_1 precedes Op_2



Op_1 is concurrent Op_2

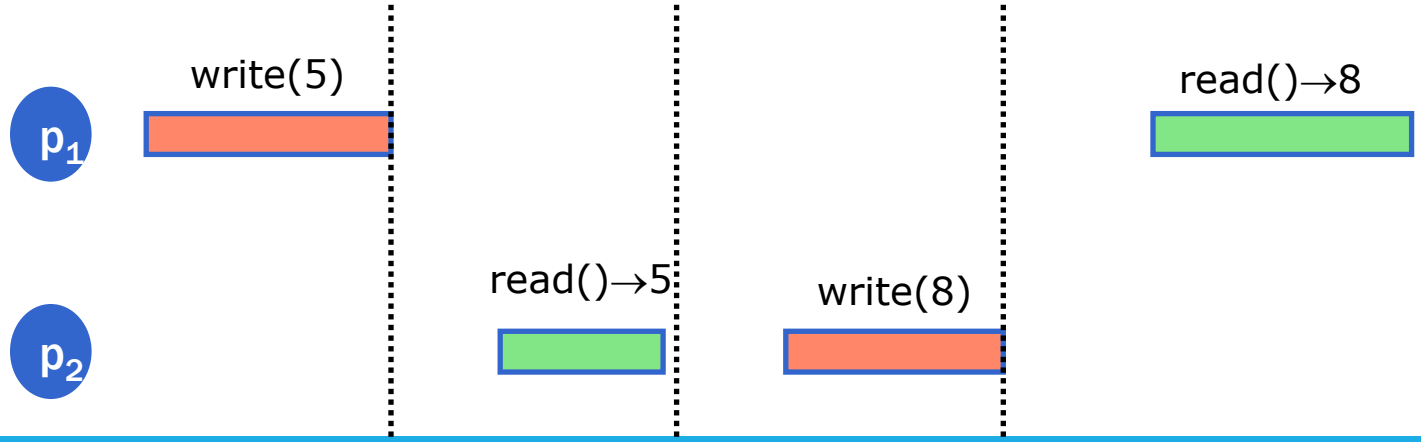
Register Semantics: Serial System, No failures

Assumptions:

- serial access: a process does not invoke an operation on the register if there is another process that previously invoked an operation on it and this latter does not yet complete
- no failures

Sequential Specification

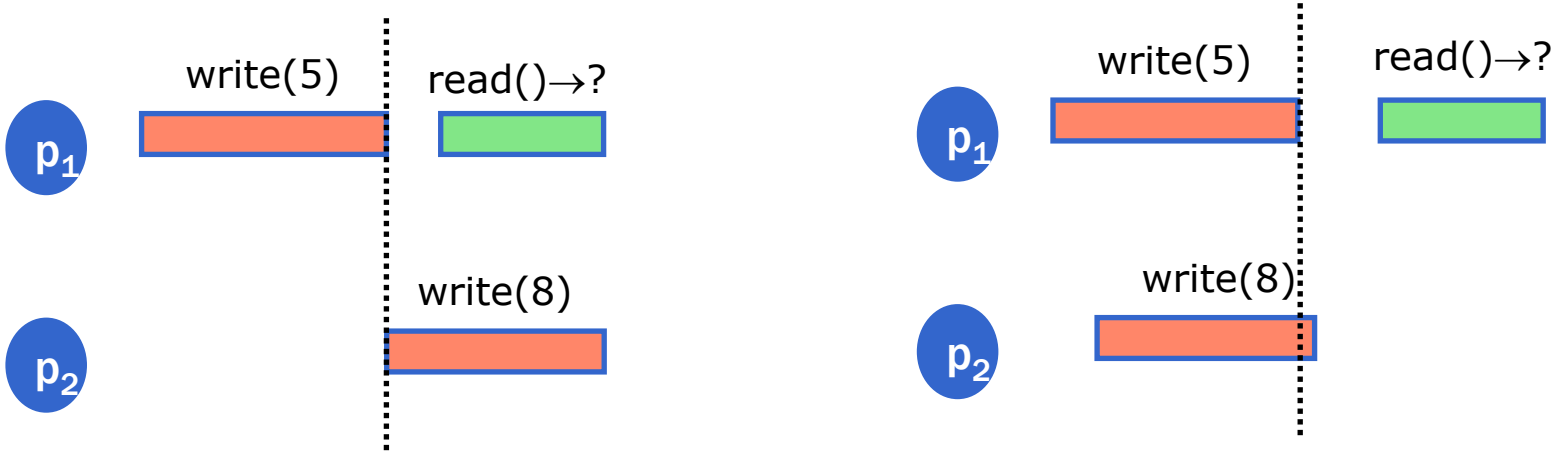
- **Liveness.** Each operation eventually terminates
- **Safety.** Each read operation returns the last value written



Register semantics: Concurrency

Assumptions:

- several processes can access the register
- concurrent access
- no failures

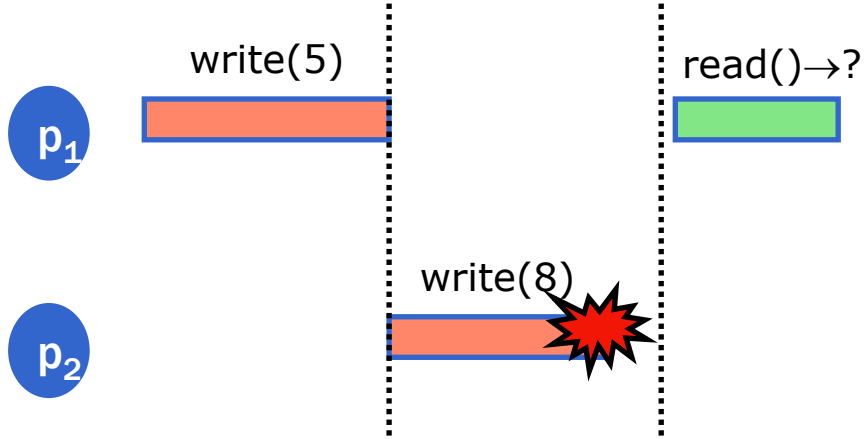


- Which value does the read operation has to return?

Register semantics: failures

Assumptions:

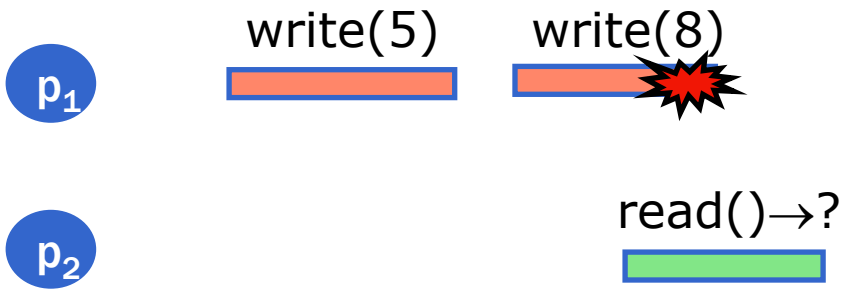
- several processes can access the register
- serial access
- processes can fail by crashing, i.e. after some point in time they stop to run their algorithm forever



Failed operation: a process fails at some time in between the invocation and the response of the operation

Which value does the read operation has to return?

Register Semantics: Concurrency & Failures



A process can invokes a write operation and then crash before the corresponding response event is generated. The write operation could have taken place or not

Register semantics: a read may return both:

- The value written by the last write operation which completes
- The value given as input to the last write operation, even though this operation will fail

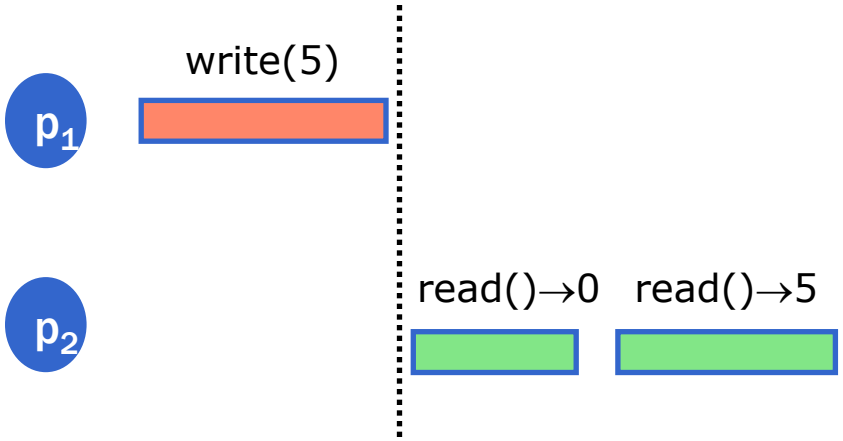
Register Specification

REGULAR AND ATOMIC

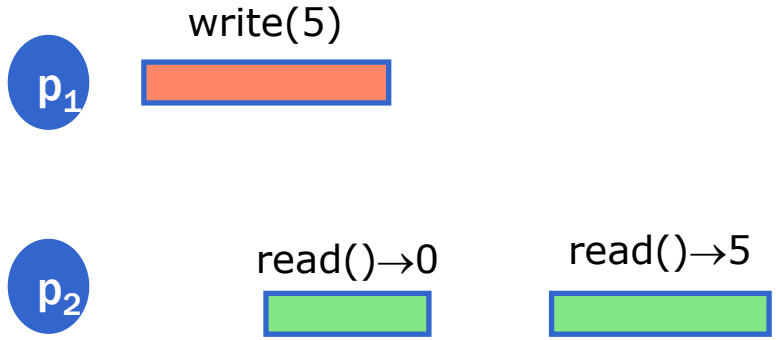
(1,N) Regular Register: Specification

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 written

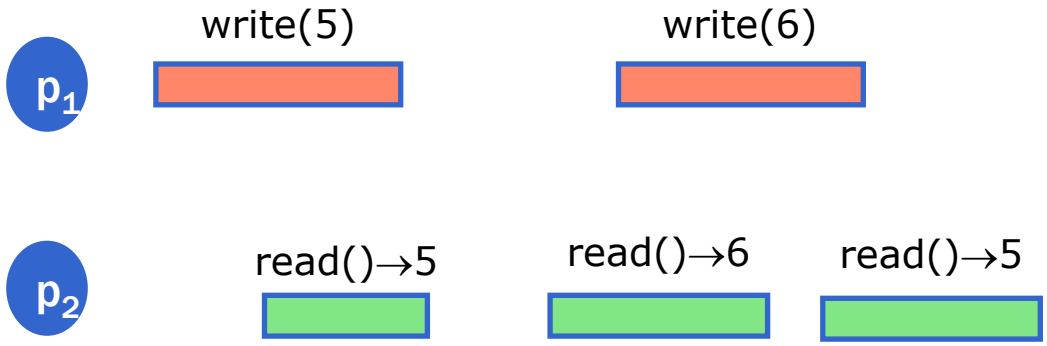


No regular



Regular

(1,N) Regular Register: Scenario



NOTE: In a regular register, a process can read a value v and then a value v', even if the writer has written v' and then v, as long as the write and the read operations are concurrent

This behavior is not allowed in an ATOMIC register

(1,N) Atomic Register: Specification

IDEA: regular register+ ordering.

Properties:

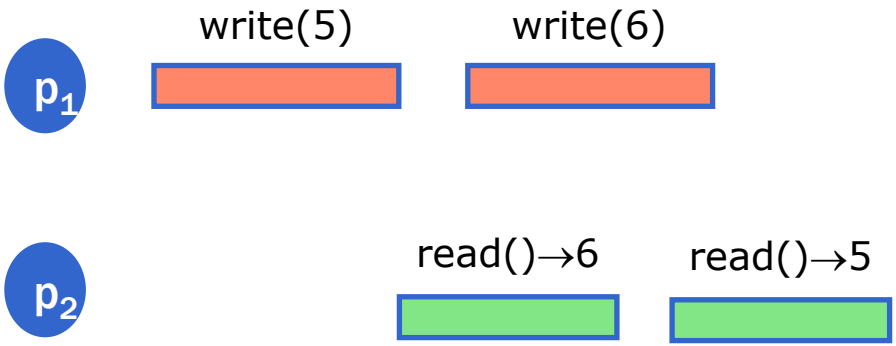
Termination. If a correct process invokes an operation, then the operation *eventually* receives the corresponding confirmation.

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

Ordering. If a read returns v_2 after a read that it precedes it has returned v_1, then v_1 cannot be written after v_2

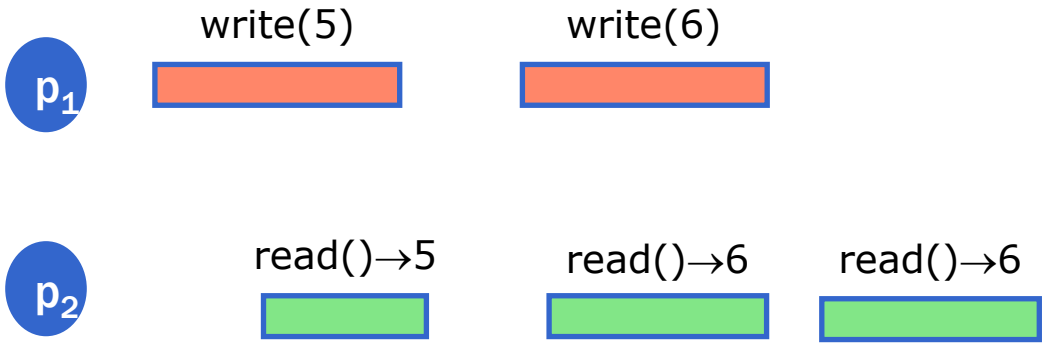
(1,N) Atomic Register: scenario

es.1



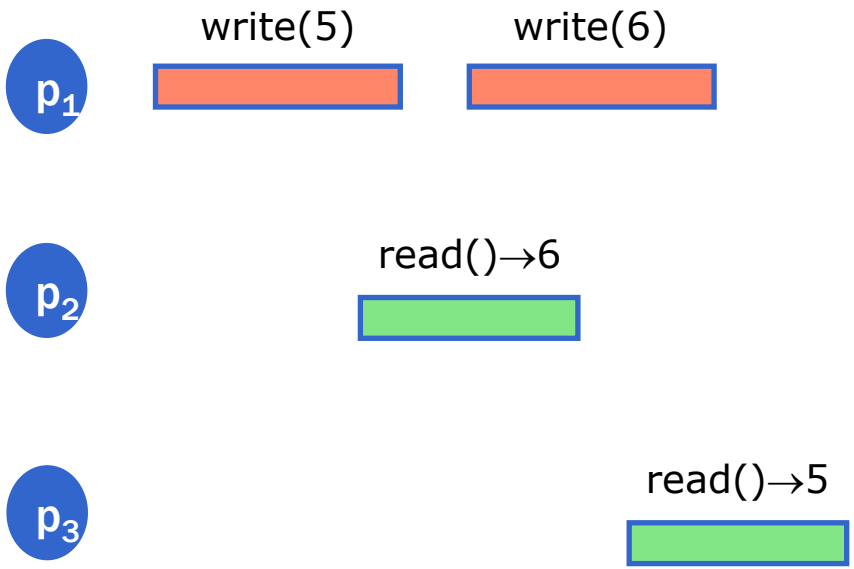
1. Regular but not atomic register: Write(5) precedes write(6). But process p_2 read first the value 6 and then the value 5

es.2



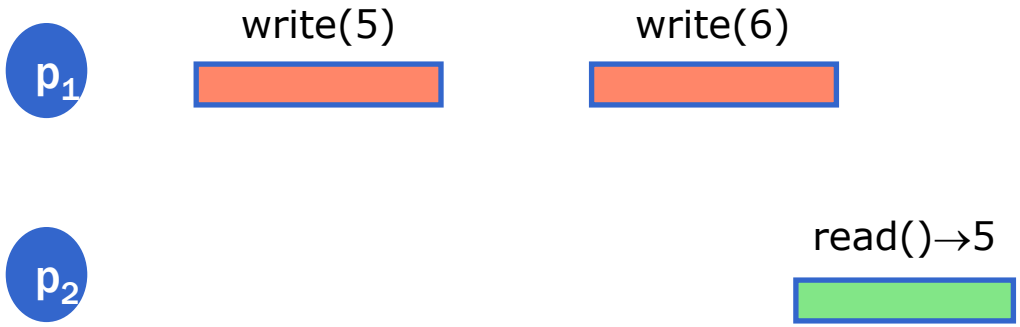
2. The register is atomic

(1,N) Atomic register: scenario



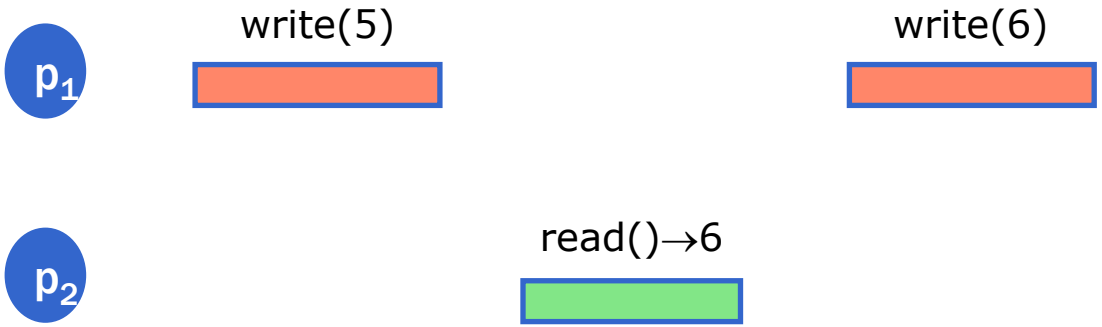
Not atomic register: the precedence relation also refers to read operations issued by different processes

Scenario 1



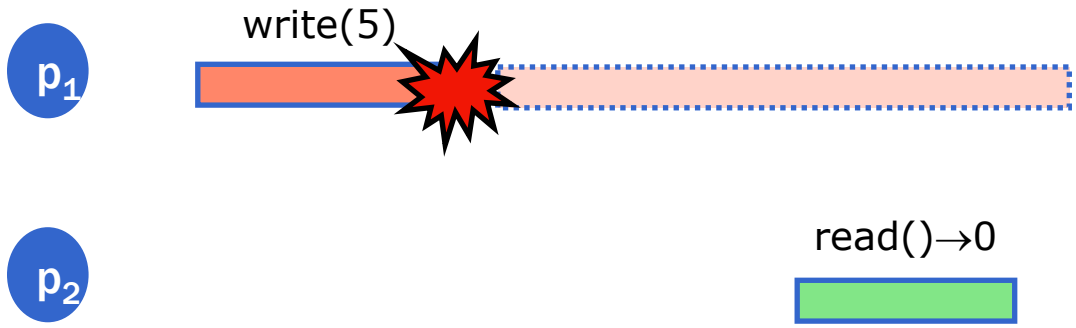
ATOMIC and REGULAR.

Scenario 2



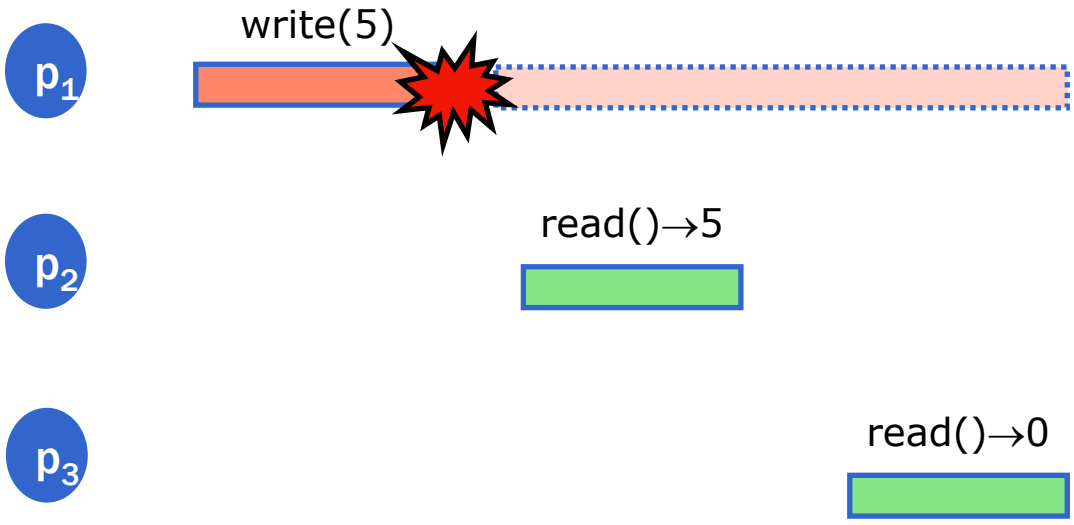
NOT ATOMIC and NOT REGULAR

Scenario 3



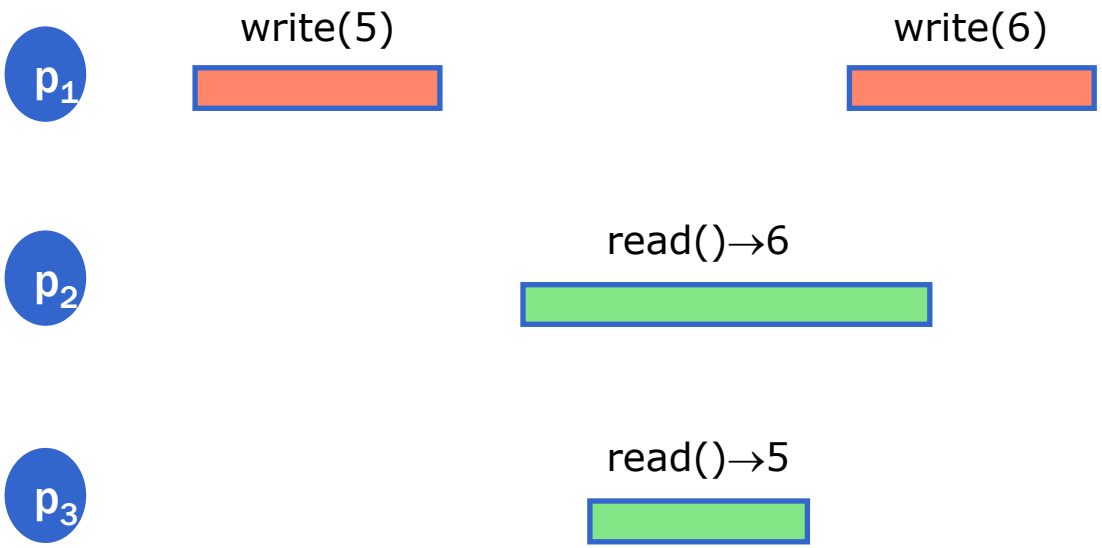
ATOMIC and so REGULAR. write(5) executed by p_1 fails. So, it does not complete and it is concurrent with the read by p_2. Validity is respected.

Scenario 4



REGULAR but non ATOMIC. The ordering property is violated.

Scenario 5



ATOMIC and REGULAR

Regular Register

IMPLEMENTATION

(1,N) Regular Register: Interface

Module 4.1: Interface and properties of a $(1, N)$ regular register

Module:

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

Events:

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

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

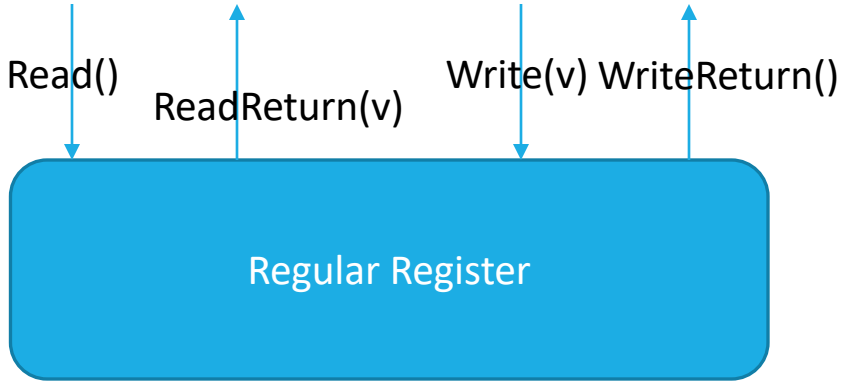
Indication: $\langle$ *onrr*, *ReadReturn* | v $\rangle$: Completes a read operation on the register with return value v.

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

Properties:

ONRR1: *Termination:* If a correct process invokes an operation, then the operation eventually completes.

ONRR2: *Validity:* A read that is not concurrent with a write returns the last value written; a read that is concurrent with a write returns the last value written or the value concurrently written.



Fail-Stop Algorithm: Read-One-Write-All Regular Register

Fail-Stop Algorithm: processes can crash but the crashes can be reliably detected by all the other processes

- failure model: crash
- perfect failure detector:
  - Strong completeness. The crash of a process is eventually detected by every correct process
  - Strong accuracy. No process is detected to have crashed until it has really crashed

Read-One-Write-All RR: Communication Primitives

Perfect point-point links:

1. Reliable delivery - Let p_i be any process that sends a message m to a process p_j. If neither p_i nor p_j crashes, then p_j eventually delivers m.
2. No duplication – No message is delivered by a process more than once.
3. No creation – If a message m is delivered by some process p_j, then m was previously sent to p_j by some process p_i.

Best-Effort Broadcast (bebBroadcast):

1. Best-effort validity –For any two processes p_i and p_j . If p_i and p_j are correct, then every message broadcast by p_i is eventually delivered by p_j.
2. No duplication- No message is delivered more than once
3. No creation- If a message m is delivered by some process p_j, then m was previously broadcast by some process p_i

Fail-Stop Algorithm, Read-One-Write-All: (1,N) Regular Register

Algorithm Idea:

- Each process stores a local copy of the register
- Read-One: each read operation returns the value stored in its local copy of the register
- Write-All: each write operation updates the value locally stored at each process the writer consider to have not crashed
- A write completes when the writer receives an ack from each process that has not crashed

(1,N) regular register

Algorithm 4.1: Read-One Write-All

Implements:

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

Uses:

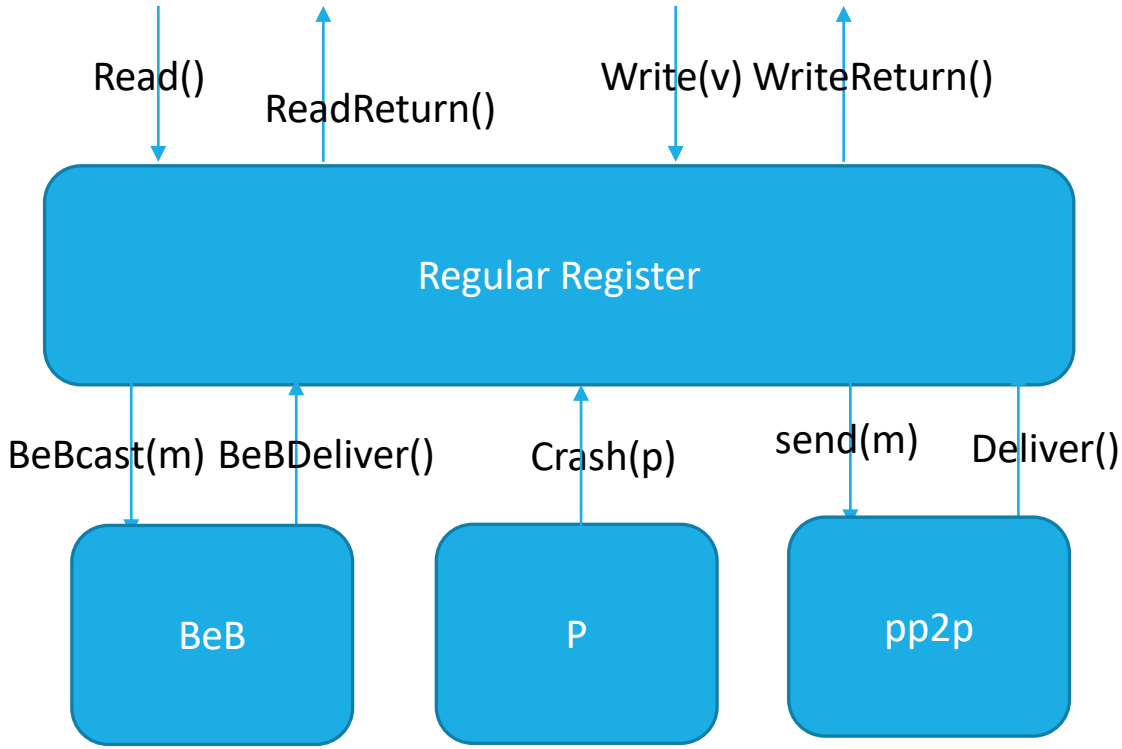
BestEffortBroadcast, **instance** *beb*;
PerfectPointToPointLinks, **instance** *pl*;
PerfectFailureDetector, **instance** $\mathcal{P}$.

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

$val := \perp$;
$correct := \Pi$;
$writeset := \emptyset$;

upon event $\langle$ $\mathcal{P}$, *Crash* | p $\rangle$ **do**

$correct := correct \setminus \{p\}$;



(1,N) regular register

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* $\rangle$;

Read() operation Implementation

Write() operation Implementation

(1,N) regular register

Correctness:

Termination –

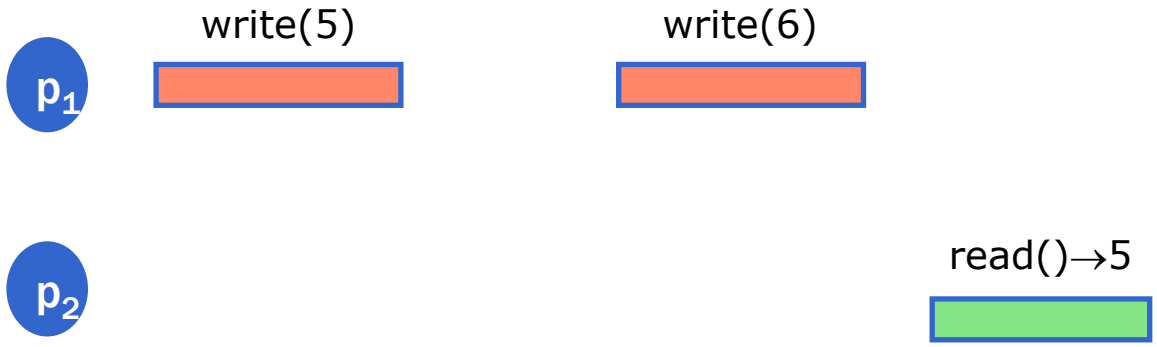
- read: trivial, it is local.
- write: from the properties of the communication primitives and from the *completeness property of the perfect failure detector*.
- *Validity* – Because of the *strong accuracy* property of the perfect failure detector, each write operation can complete only after all processes that do not crash have updated their local copy of the register. So, the two following cases can hold:
  - The read operation is not concurrent with the last write that has been invoked,the process will read the last value written
  - The read operation is concurrent with the last write. For the *no creation* property of the channels, the value returned is either the last value written or the one being written. This latter is concurrent with the read operation

Performance:

- *Write* – At most 2N messages.
- *Read* – 0 msg, it is local

Read-One-Write-All RR: problem

- The algorithm does not ensure validity if the failure detector is not perfect. The following scenario could happen:



- P_1 invokes write(6) and then falsely suspects p_2. Thus, p_1 completes the write operation without waiting for the ack of p_2, i.e. without being sure that the value 6 has been written in the local copy of the register at p_2

Fail-silent algorithm:
Majority Voting Regular Register

Fail-silent algorithm:"process crashes can never be reliably detected"

- Failure model: crash
- No perfect failure detector

Assumptions:

- N processes whose 1 writer and N readers
- A majority of correct processes

Communication Primitives:

- Perfect point-to-point link
- Best-effort broadcast

Fail-silent algorithm: Majority Voting Regular Register

IDEA:

- Each process locally stores a copy of the current value of the register
- Each written value is univocally associated to a timestamp
- The writer and the reader processes use a set of *witness processes,* to track the last value written
- *Quorum:* the intersection of any two sets of *witness processes* is not empty
- "Majority Voting": each set is constituted by a majority of processes

(1,N) regular register

Algorithm 4.2: Majority Voting Regular Register

Implements:

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

Uses:

BestEffortBroadcast, **instance** *beb*;

PerfectPointToPointLinks, **instance** *pl*.

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

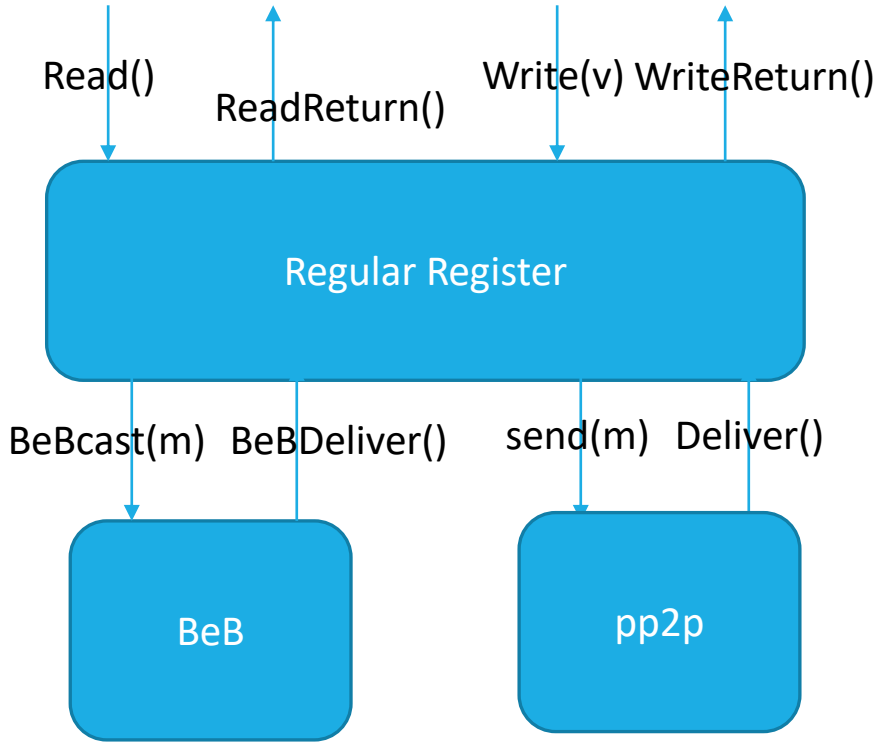
$(ts, val) := (0, \perp)$;

$wts := 0$;

$acks := 0$;

$rid := 0$;

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



(1, N) Regular Register: write() operation

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* $\rangle$;

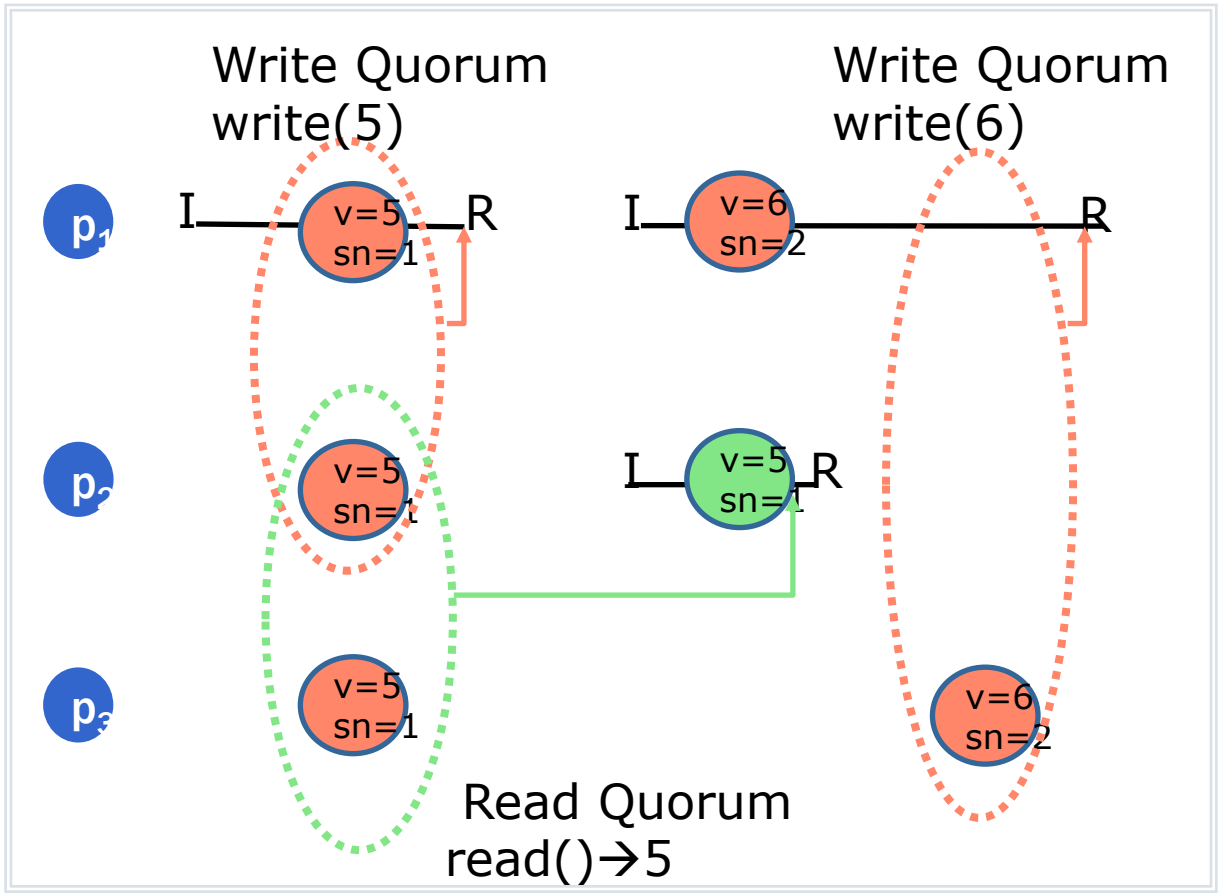
(1,N) regular register: Read() operation

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 $\rangle$;

Functioning Scenario



(1,3) regular register

- $\Pi=\{p_1,p_2,p_3\}$
- p_1 is the writer
- I=invokation
- R=response

(1,N) regular register

Correctness:

- *Termination* – from the properties of the communication primitives and the assumption of a majority of correct processes
- *Validity* – from the intersection property of the quorums

Performance:

- *Write* –at most *2N* messages
- *Read* - at most *2N* messages

Atomic Register Implementation

(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.

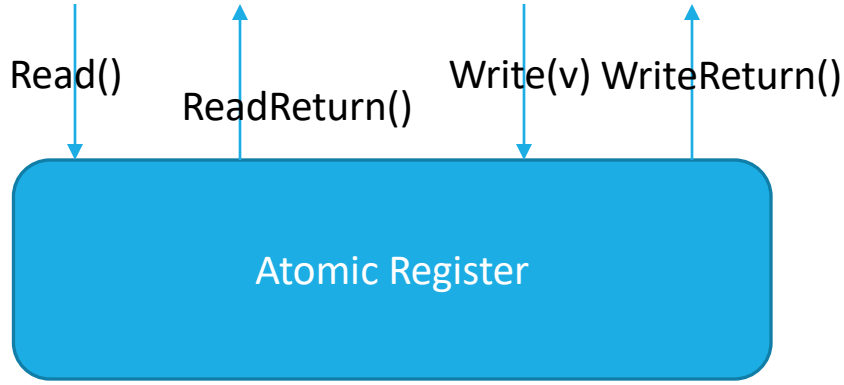
Indication: $\langle$ *onar*, *ReadReturn* $\mid v \rangle$: Completes a read operation on the register with return value v.

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.



From (1,N) Regular to (1,N) Atomic 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 ra, will be sometimes used to respectively denote regular register and atomic register

(1,N) Regular Register→(1,1) Atomic Register: PHASE 1

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

(1,N)Regular Register→ (1,1)Atomic Register

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

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

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

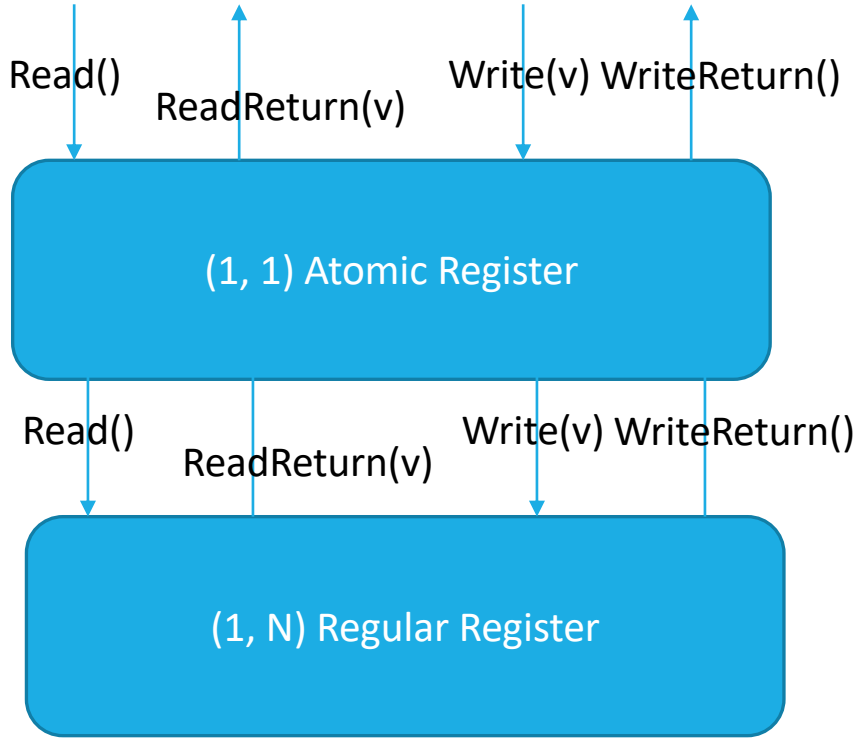
upon event $\langle$ *ooar*, *Init* $\rangle$ **do**
$(ts, val) := (0, \perp)$;
$wts := 0$;

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 $\rangle$;



(1,N) Regular Register → (1,1) Atomic Register

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 requests a write on a (1,N) regular register
- *Read* - Each read operation requests a read on a (1,N) regular register
- **NOTE:** no more msg w.r.t. (1,1) regular register implementation

(1,1)Atomic Register → (1,N) Atomic Register: Phase 2

Algorithm 4.4: From $(1, 1)$ Atomic to $(1, N)$ Atomic Registers

Implements:

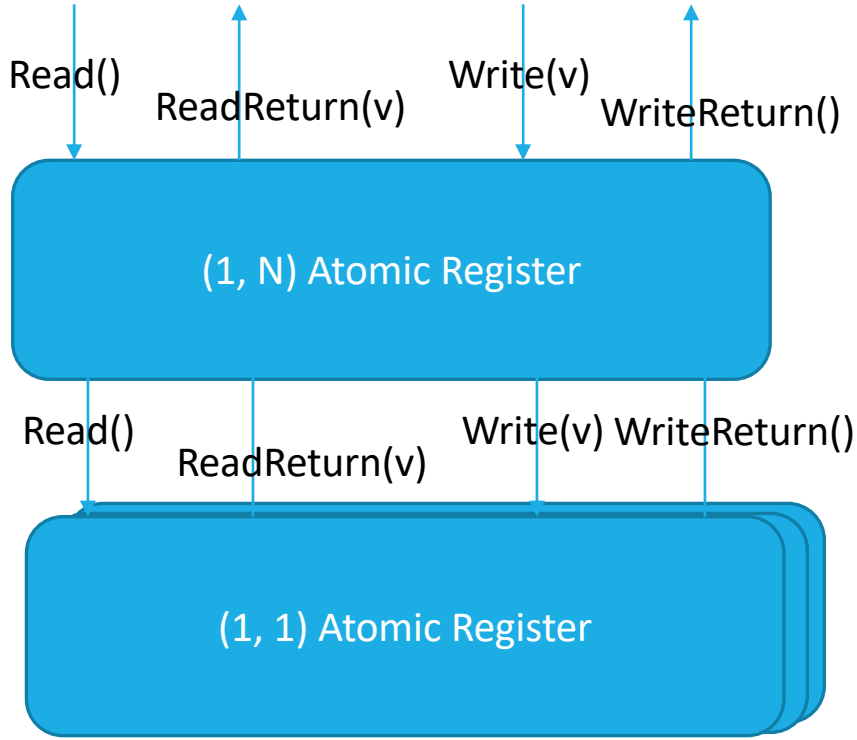
$(1, N)$-AtomicRegister, **instance** *onar*.

Uses:

$(1, 1)$-AtomicRegister (multiple instances).

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;



(1,1)Atomic Register → (1,N) Atomic Register: 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* $\rangle$;

(1,1)Atomic Register → (1,N) Atomic Register: Phase 2

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* | $(maxts, readval)$ $\rangle$;

(1,1) Atomic Register→ (1,N) Atomic Register

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 which precedes 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 s_2. Then s_1 cannot be returned.

Performance:

- *Write* – each write operation on a (1,N) atomic register requests N write operationsi on the (1,1) atomic registers.
- *Read* – Each read operation on a (1,N) atomic register requests to read N (1,1) atomic registers and to write N (1,1) atomic registers.

(1,N) Atomic Register: Fail-Stop Algorithm

Read-Impose Write-All Algorithm (1,N) Atomic Register

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 (1,N) Atomic Register

Algorithm 4.5: Read-Impose Write-All

Implements:

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

Uses:

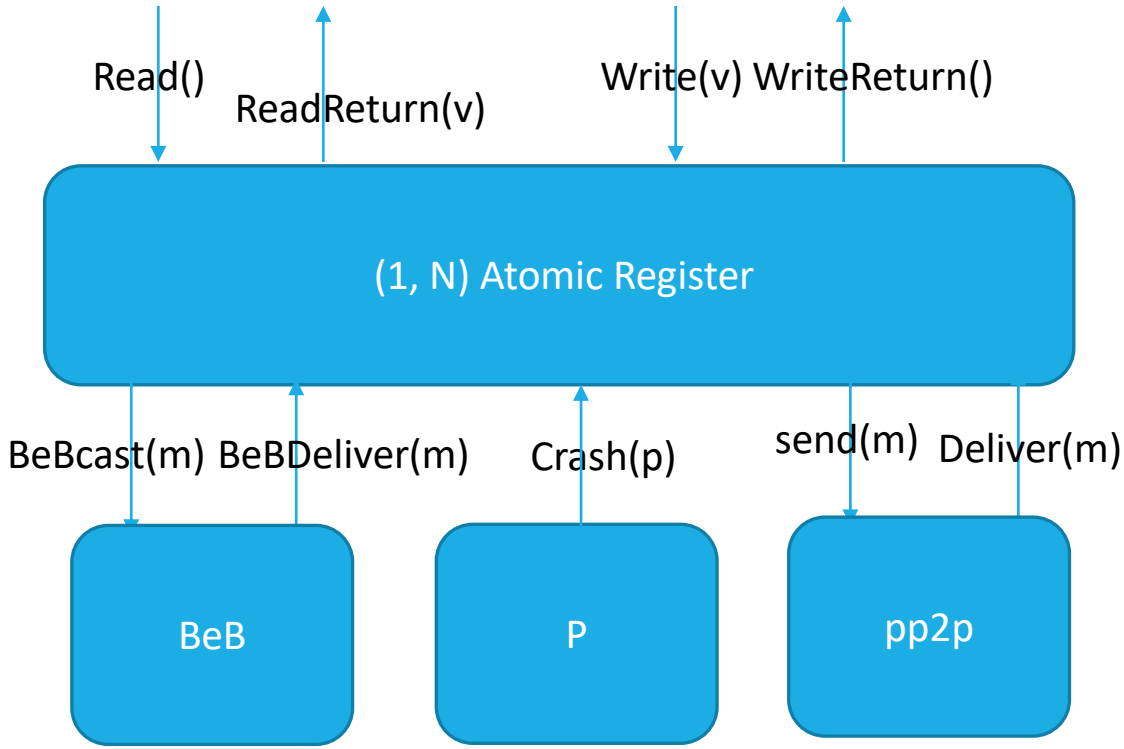
BestEffortBroadcast, **instance** *beb*;
PerfectPointToPointLinks, **instance** *pl*;
PerfectFailureDetector, **instance** $\mathcal{P}$.

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* $\setminus$ $\{p\}$;



Read-Impose Write-All (1,N) Atomic Register

```
upon event ⟨ onar, Read ⟩ do
    reading := TRUE;
    readval := val;
    trigger ⟨ beb, Broadcast | [WRITE, ts, val] ⟩;

upon event ⟨ onar, Write | v ⟩ do
    trigger ⟨ beb, Broadcast | [WRITE, ts + 1, v] ⟩;

upon event ⟨ beb, Deliver | p, [WRITE, ts', v'] ⟩ do
    if ts' > ts then
        (ts, val) := (ts', v');
    trigger ⟨ pl, Send | p, [ACK] ⟩;

upon event ⟨ pl, Deliver | p, [ACK] ⟩ then
    writeset := writeset ∪ {p};

upon correct ⊆ writeset do
    writeset := ∅;
    if reading = TRUE then
        reading := FALSE;
        trigger ⟨ onar, ReadReturn | readval ⟩;
    else
        trigger ⟨ onar, WriteReturn ⟩;
```

Read() operation Implementation

Write() operation Implementation

Read-Impose Write-All (1,N) Atomic Register

***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

(1,N) Atomic Register: Fail-Silent Algorithm

Read-Impose Write-Majority (1,N) Atomic Register

Failure model: crash

A majority of *correct processes* is assumed.

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

IDEA: A read imposes to a majority of processes to have the value read

Read-Impose Write-Majority (1,N) Atomic Register

Algorithm 4.6: Read-Impose Write-Majority (part 1, read)

Implements:

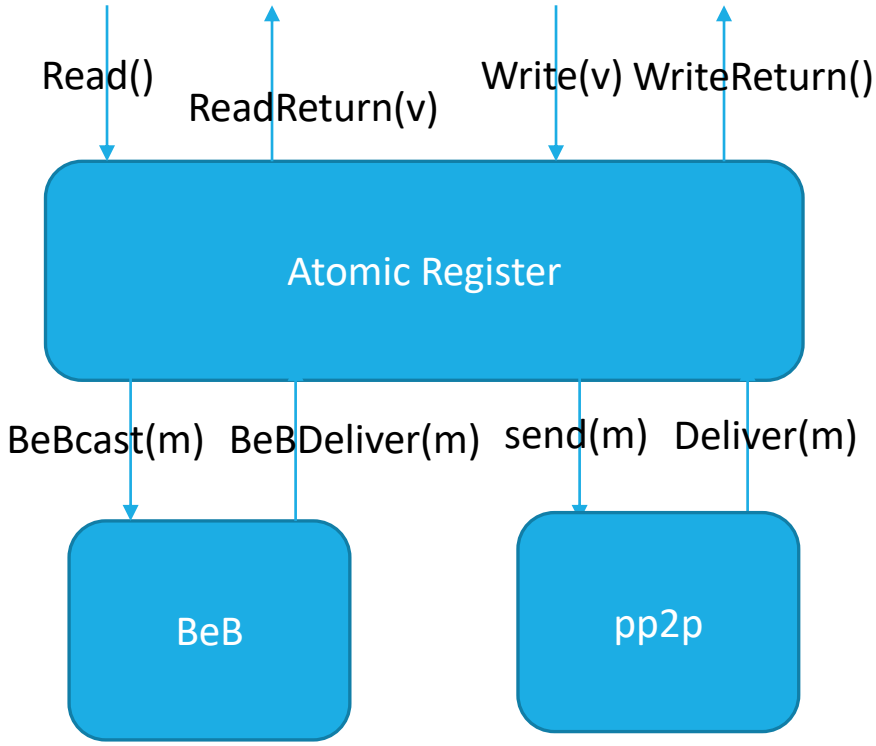
$(1, N)$-AtomicRegister, **instance** *onar*.

Uses:

BestEffortBroadcast, **instance** *beb*;
PerfectPointToPointLinks, **instance** *pl*.

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

$(ts, val) := (0, \perp)$;
$wts := 0$;
$acks := 0$;
$rid := 0$;
$readlist := [\perp]^N$;
$readval := \perp$;
$reading :=$ FALSE;



Read-Impose Write-Majority (1,N) Atomic Register

upon event $\langle$ *onar*, *Read* $\rangle$ **do**
    $rid := rid + 1$;
    $acks := 0$;
    $readlist := [\bot]^N$;
    $reading :=$ 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] $\rangle$;

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**
        $(maxts, readval) :=$ *highest*$(readlist)$;
        $readlist := [\bot]^N$;
        trigger $\langle$ *beb*, *Broadcast* | [WRITE, *rid*, *maxts*, *readval*] $\rangle$;

Read-Impose Write-Majority (1,N) Atomic Register

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 (1,N) Atomic Register

Correctness:

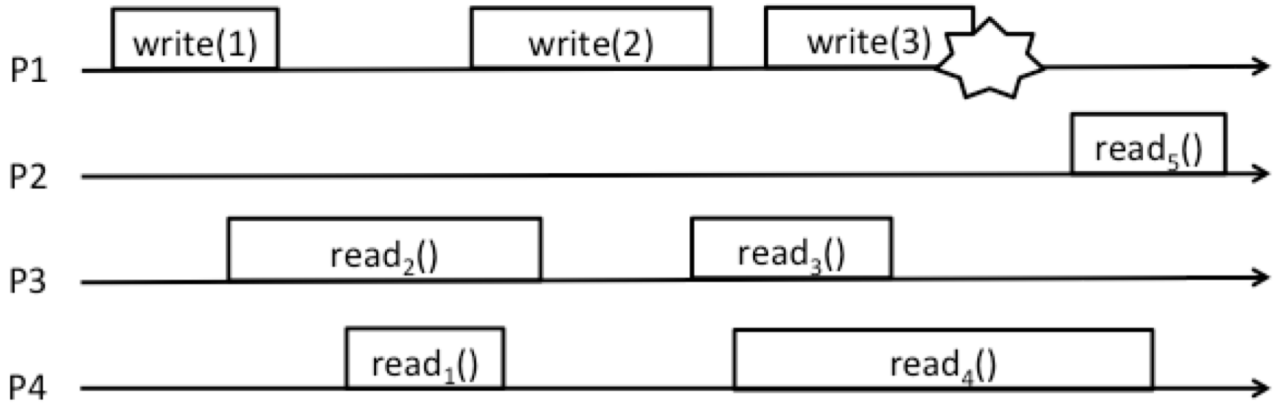
- *Termination* – as *Majority Voting (1,N) Regular Register*
- *Validity* – as *Majority Voting (1,N) Regular Register*.
- *Ordering* – due to the fact that the read imposes the write of the value read to a majority of processes and to the property of intersection of quorums.

Performance:

- *Write* – at most *2N* messages
- *Read* – at most *4N* messages

Exercise

Consider the execution depicted in the following figure and answer the questions

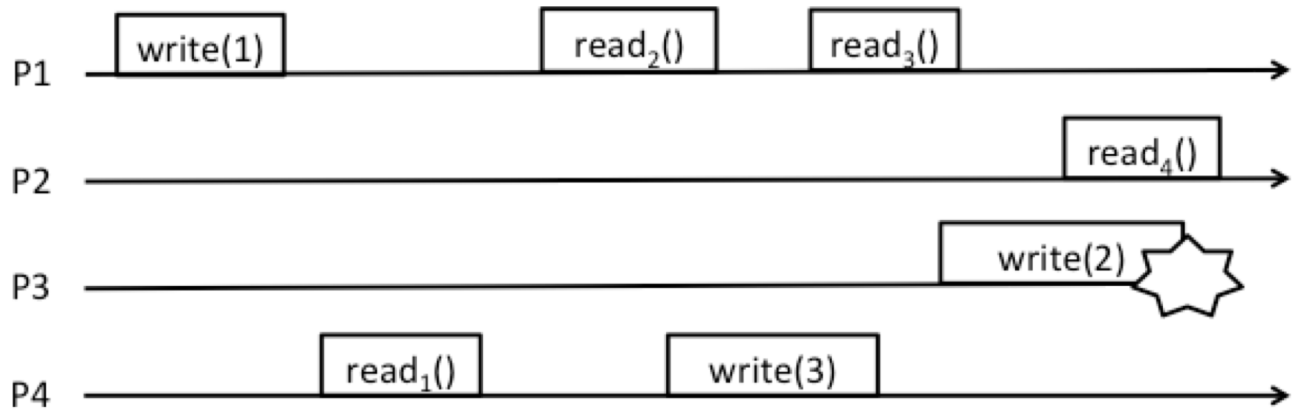


1. Define ALL the values that can be returned by read operations (Rx) assuming the run refers to a regular register.

2. Define ALL the values that can be returned by read operations (Rx) assuming the run refers to an atomic register.

Exercise

Consider the execution depicted in the following figure and answer the questions



1. Define ALL the values that can be returned by read operations (Rx) assuming the run refers to a regular register.

2. Define ALL the values that can be returned by read operations (Rx) assuming the run refers to an atomic register.

References

C. Cachin, R. Guerraoui and L. Rodrigues. Introduction to Reliable and Secure Distributed Programming, Springer, 2011

- Chapter 4 - until Section 4.3