# Distributed Algorithms Unit 2

Davide Frey, WIDE Team, Inria Rennes
davide.frey@inria.fr
https://people.irisa.fr/Davide.Frey

# Message Passing Model

- System of *n processes*
  - *process* = abstract computing unit
  - communicate by exchanging *messages on channels*

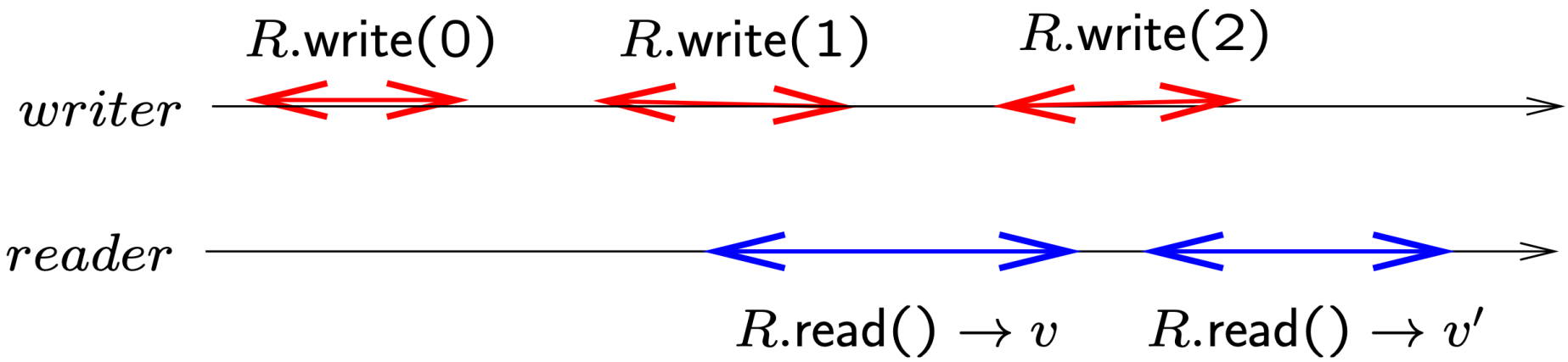
# Register Abstraction

- Basic block of a distributed memory abstraction
- Two Operations:
  - R.read() -> value
  - R.write(value)
- Will consider two variants
  - Regular
    - do not follow a sequential specification
  - Atomic
    - defined by a sequential specification

# Regular Registers

- SWMR single writer multi reader
  - only one predetermined process can write
  - anyone can read
- R.read() ->
  - if read NOT concurrent with any write, it returns
    - the current value of the register, i.e. the value that was last written
  - if read concurrent with any writes, it can return:
    - value of register before the first of these writes
    - value written by any of these writes

# Regular Register Example

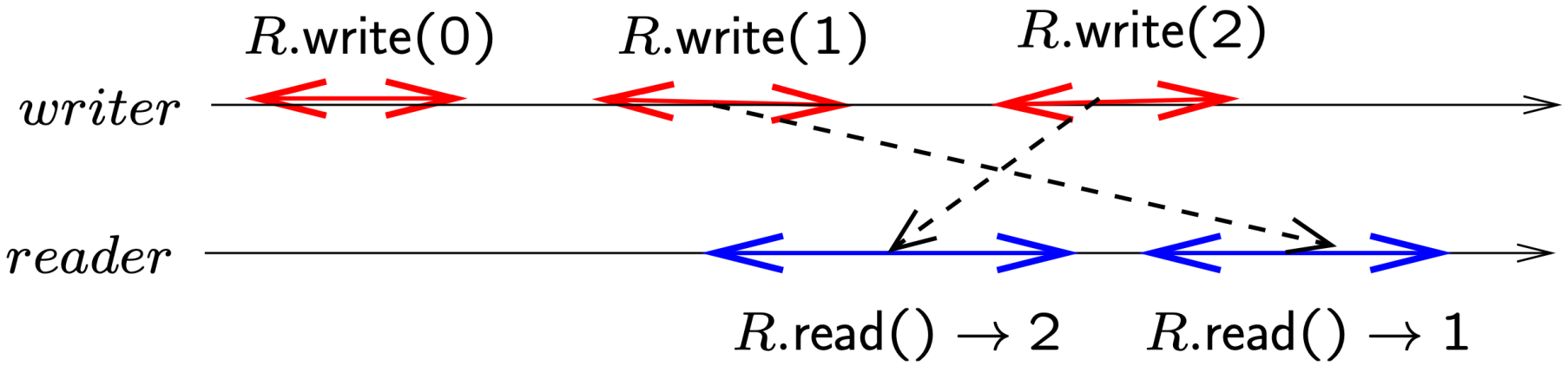



- *v can be 0, 1, or 2*
- *v' can be 1 or 2*

new/old inversion

# New/Old Inversion

- sequence returned by read operations may differ from sequence of written values
  - Write sequence 0, 1, 2
  - Read sequence v= 2, v' = 1



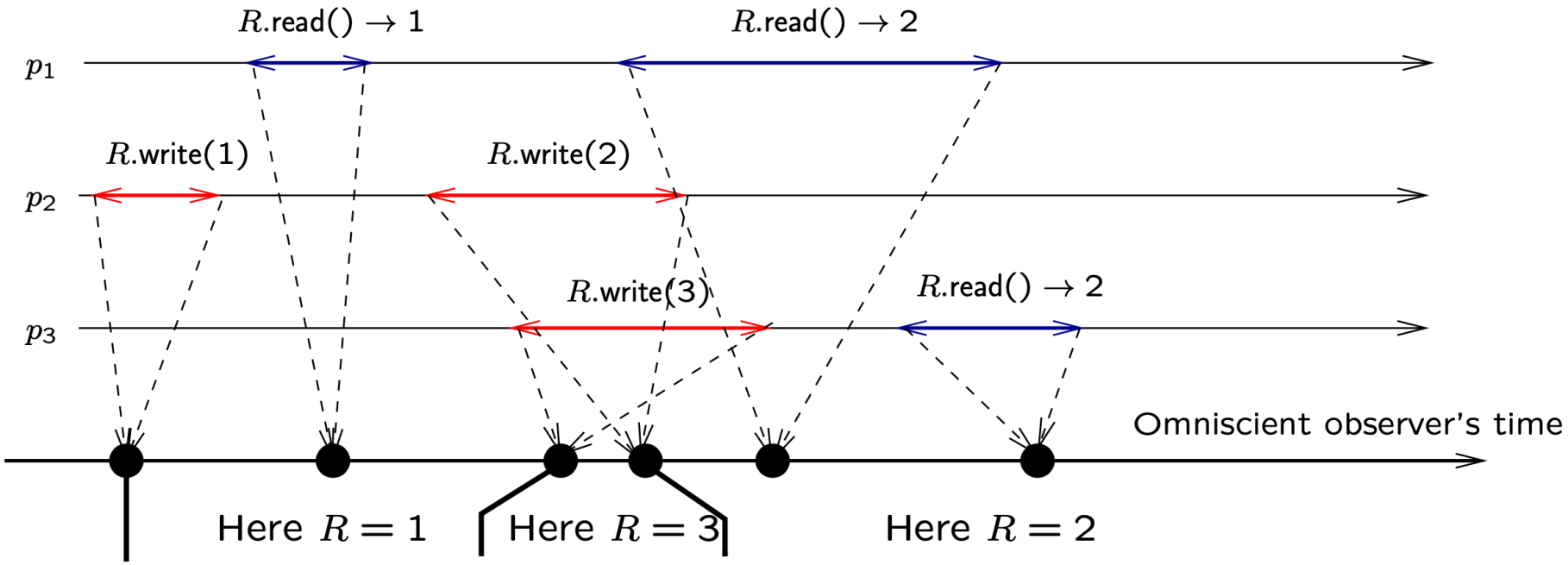

a regular register does not satisfy a sequential specification

# Atomic Register

- MWMR Multi Writer Multi Reader
- satisfies a sequential specification, i.e. no new/old inversions
- read and write operation appear as if executed in a sequence such that
  - sequence respects time order of operation (i.e. if op1 terminates before op2 starts then op1 precedes op2 in the sequence)
  - each read returns the value written by the closest preceding write in the sequence or the initial value if there is no preceding write.
- Such a sequence is called *Linearization*
- An execution can have many possible linearizations.

- Observation: a SWMR atomic register is also regular, but the converse is not true.

# Atomic Register (Example)



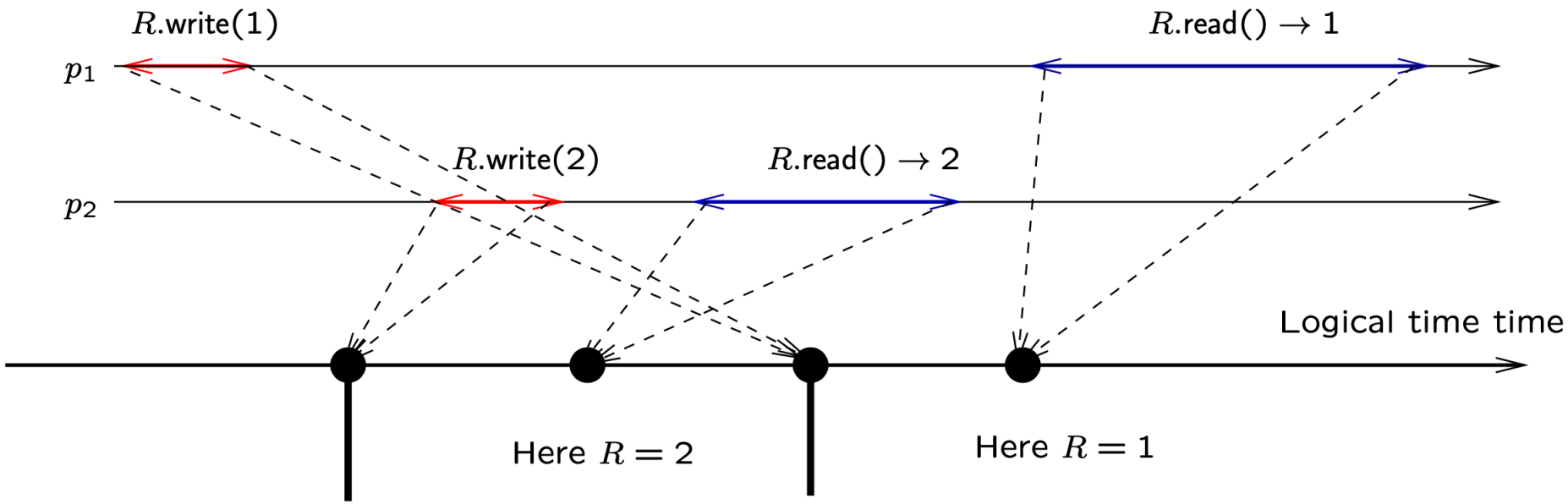

R.write$_{p2}$(1), R.read$_{p1}$()->1, R.write$_{p3}$(3), R.write$_{p2}$(2), R.read$_{p1}$()->2, R.read$_{p3}$()->2

R.write$_{p2}$(1), R.read$_{p1}$()->1, R.write$_{p2}$(2), R.write$_{p3}$(3), R.read$_{p1}$()->3, R.read$_{p3}$()->3

# Sequentially Consistent Register

- Weakened form of atomic register
- read and write operation appear as if executed in a sequence such that
  - sequence respects the process order relation (i.e. if a process invokes op1 before op2 starts then op1 op2 in the sequence)
  - each read returns the value written by the closest preceding write in the sequence or the initial value if there is no preceding write.

# Sequentially Consistent Register (Example)




R.write$_2$(2), R.read$_2$()->2, R.write$_1$(1), R.read$_1$()->1

# Composability

- Let *P* be a property defined on a set of objects
- *P* is composable if a set of objects satisfied *P* whenever each of its components satisfies *P*

Atomicity/Linearizability is composable

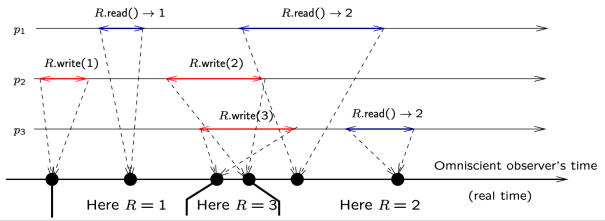
Sequential consistency is not

- Composability allows us to reason sequentially when we employ composable objects
- In practical terms, it provides *modularity*

# Do you remember these slides?

## Example: Read-Write Register

- Peer-to-Peer Model
  - All processes $\{p_1, p_2, .., p_n\}$ = P are equal.
- Distributed RW register
  - Host a copy of a memory register. Two operations: read, write
  - Should behave atomically ("one copy semantics")






## Read-Write Register (cont.)

- Fault model
  - Any number of processes may crash (up to |P|-1)
  - Message do arrive, but may take arbitrary long (asynchrony)
- Question
  - Can we implement a shared atomic RW register in this model ?

# Atomic Register in MP requires t<n/2

- n = total number of processes
- t = number of processes that can crash

- **Theorem** There is no algorithm that implements an atomic R/W register in an asynchronous system where t>=n/2 processes can crash.

- Proof by indistinguishability

# Atomic Register in MP requires t<n/2: proof by indistinguishability

assume t>=n/2

P1 P2

two partitions of size at most *ceil(n/2)*

we observe that *max (|P1|,|P2|) < t* ➔ there are executions in which all processes in P1 (or P2) crash

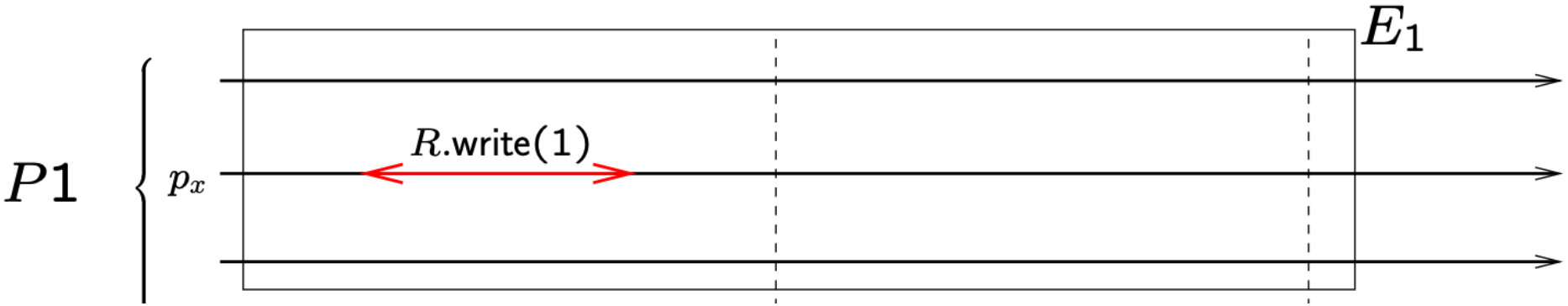
# Atomic Register in MP requires t<n/2: proof by indistinguishability

assume there is an algorithm *A* implementing atomic register R

let R's initial value be 0

Execution E1

- all processes in P2 crash, all those in P1 are correct
- a process px ∈ P1 executes R.write(1), no other process invokes any operation
- let $t_{write}$ be an a finite time after the write terminates



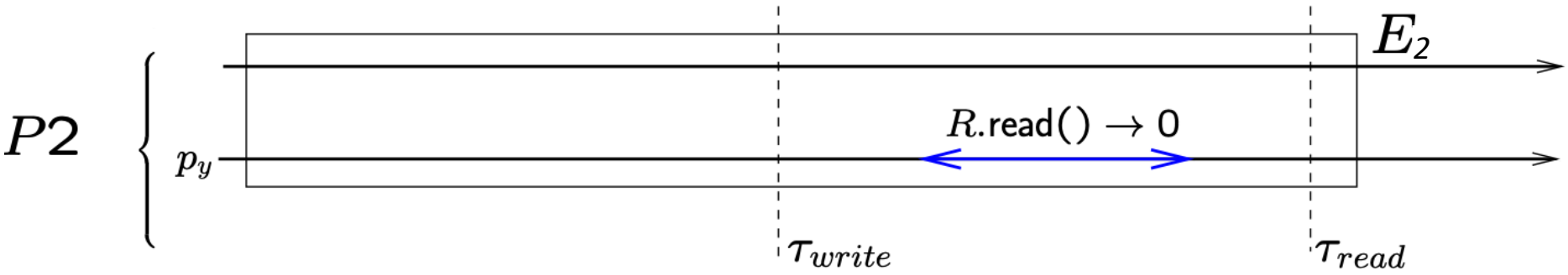
# Atomic Register in MP requires t<n/2: proof by indistinguishability

assume there is an algorithm *A* implementing atomic register R

let R's initial value be 0

Execution E2

- all processes in P1 crash, all those in P2 are correct
- processes in P2 do nothing until $t_{write}$
- after $t_{write,}$ py ∈ P2 issues R.read()->0, no other process executes any operation
- let $t_{read}$ be a finite time after the read operation terminates



# Atomic Register in MP requires t<n/2: proof by indistinguishability

assume there is an algorithm *A* implementing atomic register R

let R's initial value be 0

Execution E1

- all processes in P2 crash, all those in P1 are correct
- a process px ∈ P1 executes R.write(1), no other process invokes any operation
- let $t_{write}$ be an a finite time after the write terminates

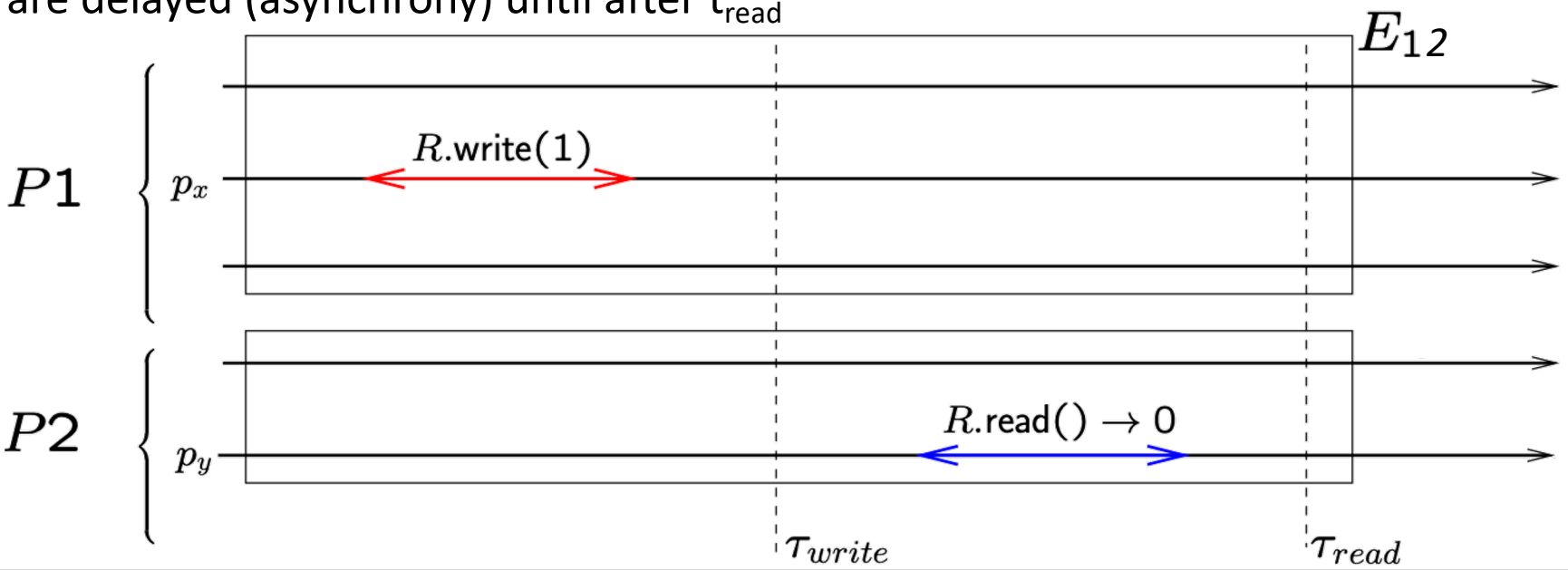
Execution E2

- all processes in P1 crash, all those in P2 are correct
- processes in P2 do nothing until $t_{write}$
- after $t_{write,}$ py ∈ P2 issues R.read()->0, no other process executes any operation
- let $t_{read}$ be a finite time after the read operation terminates

# Atomic Register in MP requires t<n/2: proof by indistinguishability

Execution E12

- No process crashes
- E12 is the same as E1 until $t_{write}$ (except for the crashes)
- E12 is the same as E2 after $t_{write}$ and until $t_{read}$
- The messages sent by processes in P1 to processes in P2 and those from P2 to P1 are delayed (asynchrony) until after $t_{read}$



# Atomic Register in MP requires t<n/2: proof by indistinguishability

Execution E12

- No process crashes
- E12 is the same as E1 until $t_{write}$ (except for the crashes)
- E12 is the same as E2 after $t_{write}$ and until $t_{read}$
- The messages sent by processes in P1 to processes in P2 and those from P2 to P1 are delayed (asynchrony) until after $t_{read}$

Process py cannot distinguish E12 from E2 until $t_{read}$ so its read must return 0
But by atomicity its read should return 1 in E12

Contradiction. Hence algorithm A cannot exist

Inria

# Implementing a Register

**operation** $R$.write $(v)$ **is**
% This code is for the single writer $p_w$ %
$wsn_w \leftarrow wsn_w + 1$;
broadcast write $(v, wsn_w)$;
wait (ack_write $(wsn_w)$ rec. from a majority of proc.);
return ().

**when** write $(val, wsn)$ **is received by** $p_i$ **from** $p_w$ **do**
**if** $(wsn \geq wsn_i)$ **then** $reg_i \leftarrow val$; $wsn_i \leftarrow wsn$ **end if**;
send ack_write $(wsn)$ to $p_w$.

Inria

# Implementing a Register

**operation** $REG$.read () **is** % This code is for any $p_i$ %
$reqsn_i \leftarrow reqsn_i + 1$;
broadcast read_req $(reqsn_i)$;
wait (ack_read_req $(reqsn_i, -, -)$ received
from a majority of proc.);
**let** ack_read_req $(reqsn_i, -, v)$ **be** a message
received with the greatest write sequence nb;
return $(v)$.

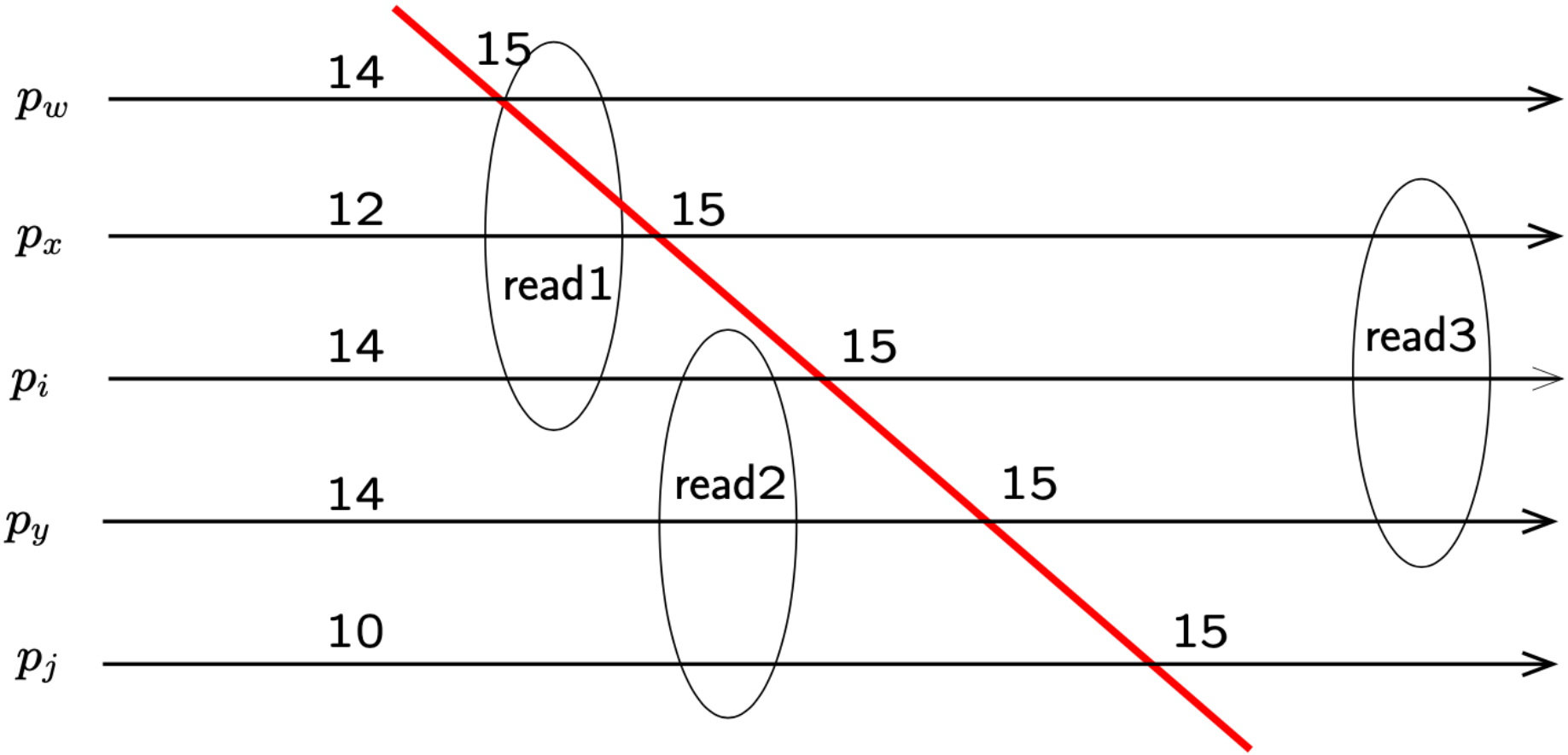
**when** read_req $(rsn)$ **is received from** $p_j$ **do**
send ack_read_req $(rsn, wsn_i, reg_i)$ to $p_j$.

Inria

# Implementing a Register

- What register does the above algorithm implement?
  - Is it regular? Why?
  - Is it Atomic? Why?

# Not an Atomic Register



# From Regular to Atomic Register

- Read operation should write back its value
- this guarantees that the value returned by a read is known by a majority

**operation** $R$.read () **is**
$reqsn_i \leftarrow reqsn_i + 1$;
broadcast read_req $(reqsn_i)$;
wait (ack_read_req $(reqsn_i, -, -)$ received
from a majority of proc.);
**let** ack_read_req $(reqsn_i, msn, v)$ **be** a message received
with the greatest write sequence nb $msn$;
broadcast write$(v, msn)$;
wait (ack_write $(msn)$ rec. from a majority of proc.);
return $(v)$.

# From Regular to Atomic Register

- Server side

**when** write $(val, wsn)$ **is received by** $p_i$ **from** $p_j$ **do**
    **if** $(wsn \geq wsn_i)$ **then** $reg_i \leftarrow val$; $wsn_i \leftarrow wsn$ **end if**;
    send ack_write $(wsn)$ to $p_j$.

# From Atomic SWMR to Atomic MWMR

- Need a global sequence number to totally order operations
- Lamport's logical clocks

# Lamport's Logical Clocks

[L. Lamport. "Time, clocks, and the ordering of events in a distributed system". Communications of the ACM, 21(7):558-565, July 1978]

- Define logical timestamps for Message Passing systems
- *Key concept: happens-before* relation $e \rightarrow e'$
  - If events *e* and *e'* occur in the same process and *e* occurs before *e'*, then $e \rightarrow e'$
  - If *e=send(msg)* and *e'=recv(msg)*, then $e \rightarrow e'$
  - $\rightarrow$ is transitive
- If neither $e \rightarrow e'$ nor $e' \rightarrow e$, they are concurrent ($e || e'$)

Inria

# Lamport's Logical Clocks

- Define logical timestamps for Message Passing systems [Lamport 1978]
- *happens-before* relation $e \rightarrow e'$:
  - If events $e$ and $e'$ occur in the same process and $e$ occurs before $e'$, then $e \rightarrow e'$
  - If $e$=*send(msg)* and $e'$=*recv(msg)*, then $e \rightarrow e'$
  - $\rightarrow$ is transitive

- Replace unidimensional sequence numbers by two dimensional timestamps

# Lamport's Logical Clocks

[L. Lamport. "Time, clocks, and the ordering of events in a distributed system". Communications of the ACM, 21(7):558-565, July 1978]

- The happens-before relationship captures *potential causal ordering* among events
  - Two events can be related by the happens-before relationship even if there is no real (causal) connection among them
  - Also, since information can flow in ways other than message passing, two events may be causally related even neither of them happens-before the other

# Lamport's Logical Clocks
# Scalar Clocks

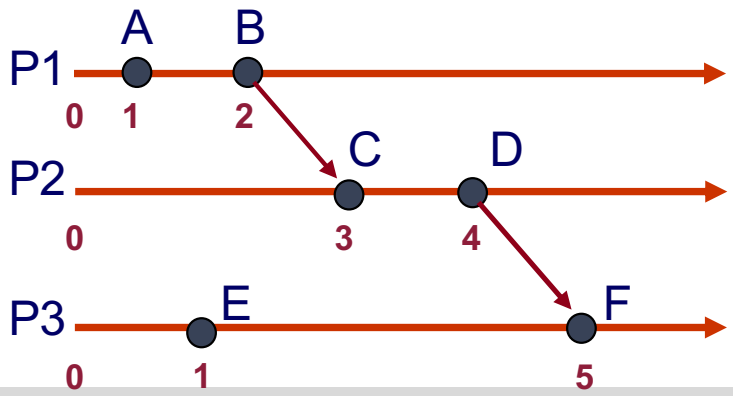
- Lamport's simple mechanism to capture happens-before
  - Scalar Clocks
  - Integers to represent the clock value
  - No relationship with a physical clock whatsoever

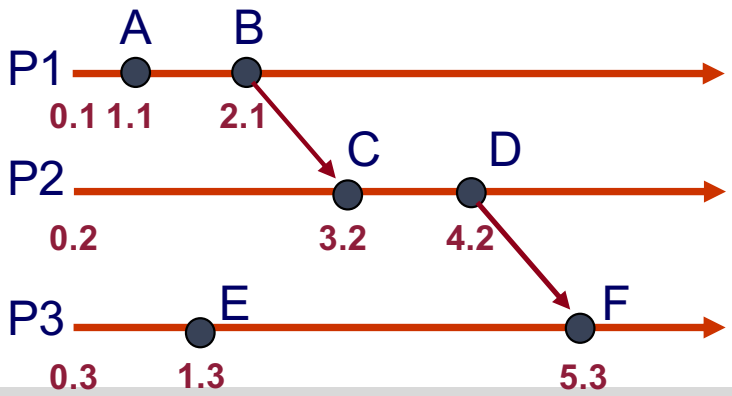
- Each process $p_i$ keeps a logical scalar clock $L_i$
  - $L_i$ starts at zero
  - $L_i$ is incremented before $p_i$ sends a message
  - Each message sent by $p_i$ is timestamped with $L_i$
  - Upon receipt of a message, $p_i$ sets $L_i$ to: MAX(msg timestamp, $L_i$) + 1
- Can show that:
  $e \rightarrow e' \Rightarrow L(e) < L(e')$

# Lamport's Logical Clocks
# From Scalar Clocks to Timestamps

- Scalar Clocks provide a partial ordering.
- To achieve total ordering, attach process IDs
  <L, i>
- Sort timestamp by lexicographical total order

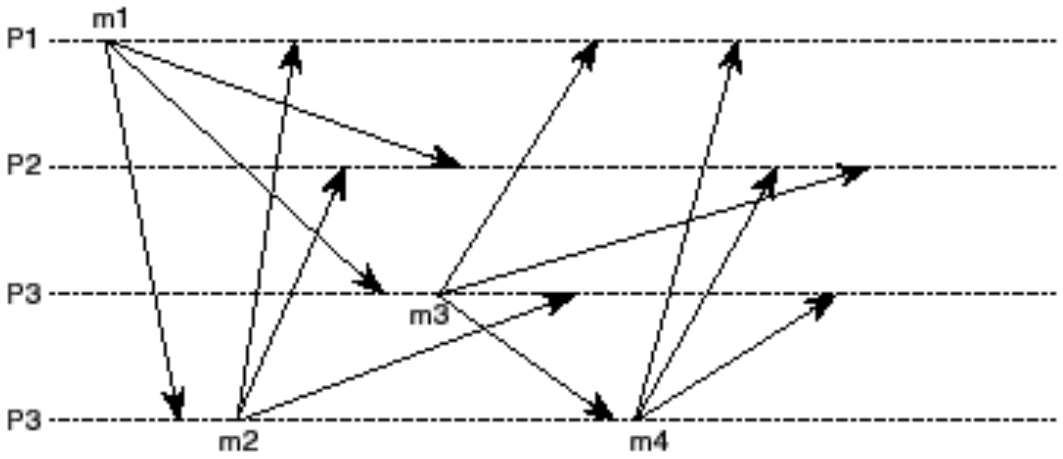
$$\langle \ell c1, i \rangle < \langle \ell c2, j \rangle \equiv \big((\ell c1 < \ell c2) \vee (\ell c1 = \ell c2 \wedge i < j)\big)$$



# Exercise

- Consider 4 processes exchanging messages as in figure:




Which is the value of Lamport's clocks at the end of the reported period?

# Lamport Timestamps: Summary

- Provide total ordering among events
  - In Lamport's example above
    - send Event
    - receive Event
  - In our MultiWriterMultiReader
    - write Event

# Complete ABD Algorithm (1/3)

**operation** REG.write $(v)$ **is**

(1) $reqsn_i \leftarrow reqsn_i + 1$;

% Phase 1: acquire information on the system state %

(2) broadcast WRITE_REQ $(reqsn_i)$;

(3) wait(ACK_WRITE_REQ $(reqsn_i, -)$ received from a majority of processes);

(4) **let** $msn$ **be** the greatest sequence number previously received in an ACK_WRITE_REQ $(reqsn_i, -)$ message;

% Phase 2 : update system state %

(5) broadcast WRITE $(reqsn_i, v, msn + 1, i)$;

(6) wait (ACK_WRITE $(reqsn_i)$ received from a majority of processes);

(7) return().

Inria

# Complete ABD Algorithm (2/3)

**operation** REG.read () **is**
(8) $reqsn_i \leftarrow reqsn_i + 1$;
% Phase 1: acquire information on the system state %
(9) **broadcast** READ_REQ $(reqsn_i)$;
(10) **wait** ( ACK_READ_REQ $(reqsn_i, -, -, -)$ received from a majority of processes);
(11) **let** $\langle msn, mlw \rangle$ **be** the greatest timestamp received in
an ACK_READ_REQ $(reqsn_i, -, -, -)$ message;
(12) **let** $v$ **be** such that ACK_READ_REQ $(req\_sn_i, msn, mlw, v)$ has been received;
% Phase 2 : update system state %
(13) **broadcast** WRITE $(reqsn_i, v, msn, mlw)$;
(14) **wait** (ACK_WRITE $(reqsn_i)$ received from a majority of processes);
(15) **return** $(v)$.

# Complete ABD Algorithm (3/3)

**when** WRITE $(rsn, val, wsn, lw)$ **is received from** $p_j$ **do** $\quad \%\ j \in \{1, \ldots, n\}\ \%$
(16) **if** $\langle wsn, lw \rangle \geq (wsn_i, lw_i)$ **then** $reg_i \leftarrow val$; $wsn_i \leftarrow wsn$; $lw_i \leftarrow lw$ **end if**;
(17) send ACK_WRITE $(rsn)$ to $p_j$.

**when** READ_REQ $(rsn)$ **is received from** $p_j$ **do** $\quad \%\ j \in \{1, \ldots, n\}\ \%$
(18) send ACK_READ_REQ $(rsn, wsn_i, lw_i, reg_i)$ to $p_j$.

**when** WRITE_REQ $(rsn)$ **is received from** $p_j$ **do** $\quad \%\ j \in \{1, \ldots, n\}\ \%$
(19) send ACK_WRITE_REQ $(rsn, wsn_i)$ to $p_j$.

Inria