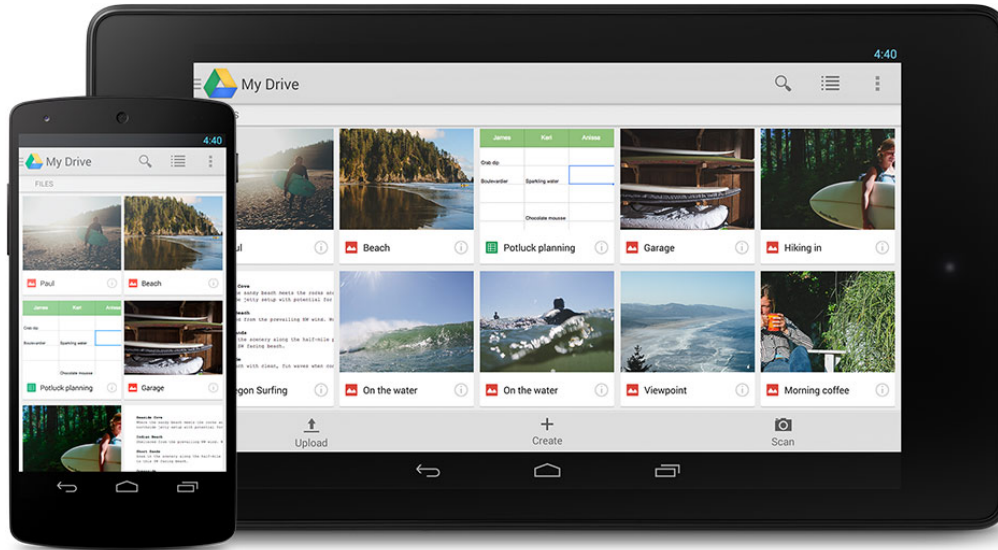


# Large-scale Data Systems

Lecture 4: Shared memory

Prof. Gilles Louppe  
[g.louppe@uliege.be](mailto:g.louppe@uliege.be)

# Today



*Towards a distributed file system.*

- How do you **share resources** between processes?
- Can we build the **illusion of single storage**?
  - While replicating data for **fault-tolerance** and **scalability**?
  - While maintaining **consistency**?

Data science  
Machine Learning  
Visualization

Data systems

Distributed systems

Registers

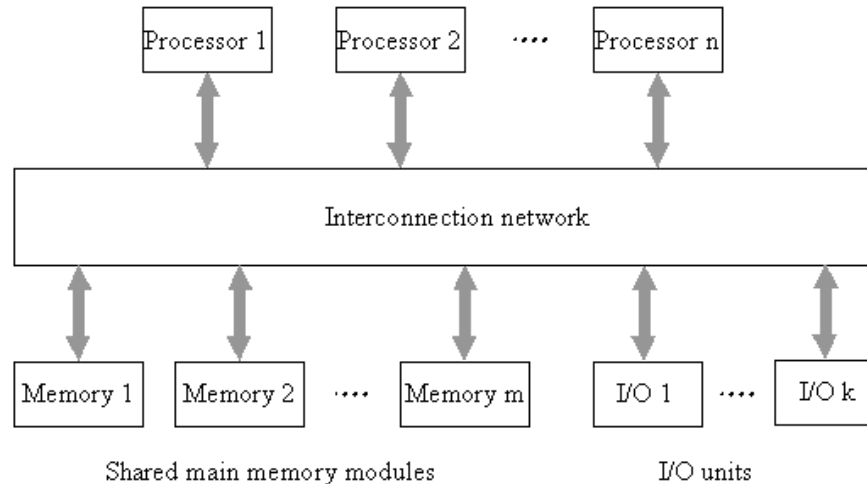
Reliable broadcast

Basic abstractions:  
Processes, links, time

Operating systems

Computer networks

# Shared memory



In a multiprocessor machine, processors typically communicate through **shared memory** provided at the hardware level (e.g., shared blocks of RAM that can be accessed by distinct CPUs).

- Shared memory can be viewed as an **array of registers** to which processors can **read** or **write**.
- Shared memory systems are **easy to program** since all processors share a single view of the data.

## Shared memory emulation

We want to **simulate** a **shared memory abstraction** in a distributed system, on top of message passing communication.

- Enable shared memory algorithms without being aware that processes are actually communicating by exchanging messages.
  - This is often much easier to program.
- Equivalent to **consistent data replication** across nodes.

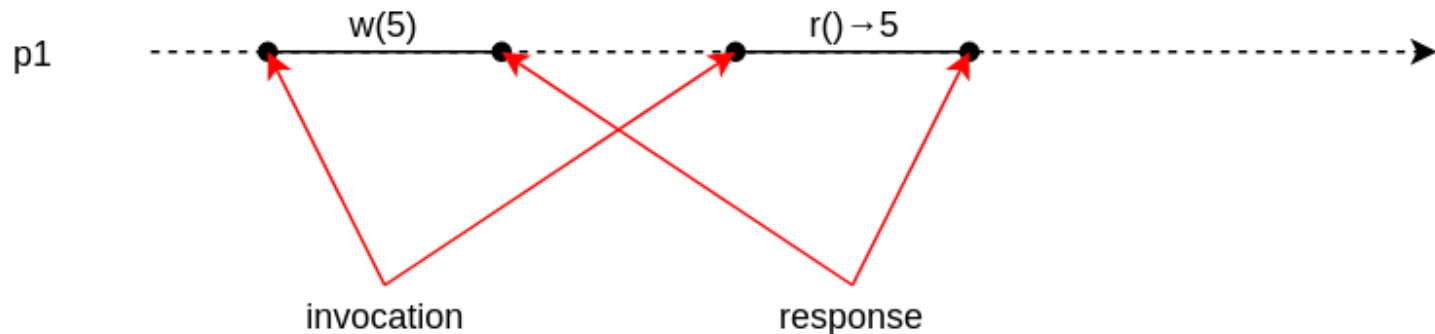
## Data replication

- Shared data allows to:
  - Reduce network traffic
  - Promote increased parallelism
  - Be robust against failures
  - Result in fewer page faults
- Applications:
  - distributed databases
  - distributed file systems
  - distributed cache
- Challenges:
  - Consistency in presence of [failures](#).
  - Consistency in presence of [concurrency](#).

# Regular registers

# Read/Write registers

- A **register** represents each memory location.
- A register contains only positive integers and is initialized to 0.
- Registers have two **operations**:
  - **read()**: return the current value of the register.
  - **write(*v*)**: update the register to value *v*.
- An operation is **not instantaneous**:
  - It is first **invoked** by the calling process.
  - It computes for some time.
  - It returns a **response** upon completion.





## Definitions

- In an execution, an operation is
  - **completed** if both invocation and response occurred.
  - **failed** if invoked but not no response was received.
- Operation  $o_1$  **precedes**  $o_2$  if response of  $o_1$  precedes the invocation of  $o_2$ .
- Operations  $o_1$  and  $o_2$  are **concurrent** if neither precedes the other.
- $(1, N)$ -register: 1 designated writer,  $N$  readers.
- $(M, N)$ -register:  $M$  writers,  $N$  readers.

# Regular registers (*onrr*)

## Module:

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

## Events:

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

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

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

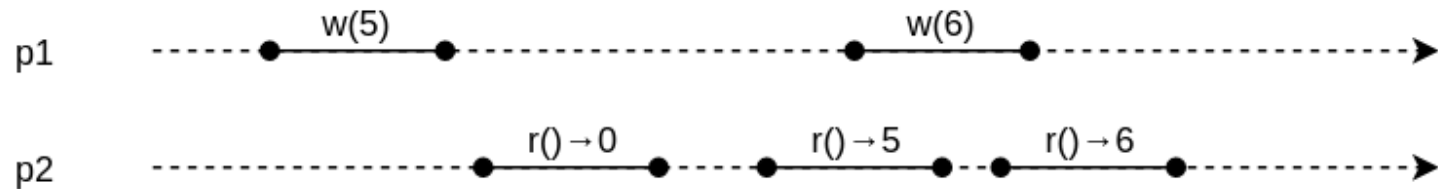
**Indication:**  $\langle \textit{onrr}, \textit{WriteReturn} \rangle$ : Completes a write operation on the register.

## 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.

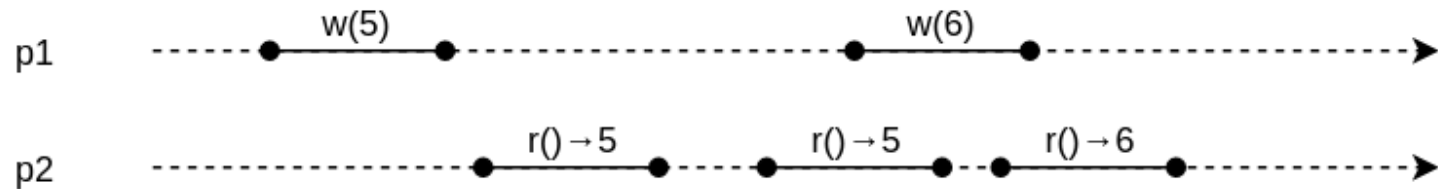
## Regular register example (1)



### Exercise

Regular or non-regular?

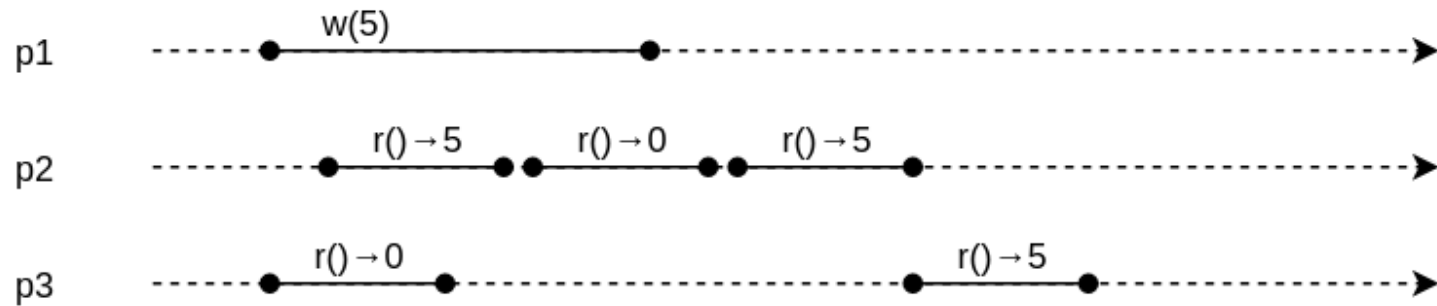
## Regular register example (2)



### Exercise

Regular or non-regular?

## Regular register example (3)



### Exercise

Regular or non-regular?

## Centralized algorithm

- Designates one process as the **leader**.
  - E.g., using the leader election abstraction (see Lecture 2).
- To **read()**:
  - Ask the leader for latest value.
- To **write( $v$ )**:
  - Update leader's value to  $v$ .

### Exercise

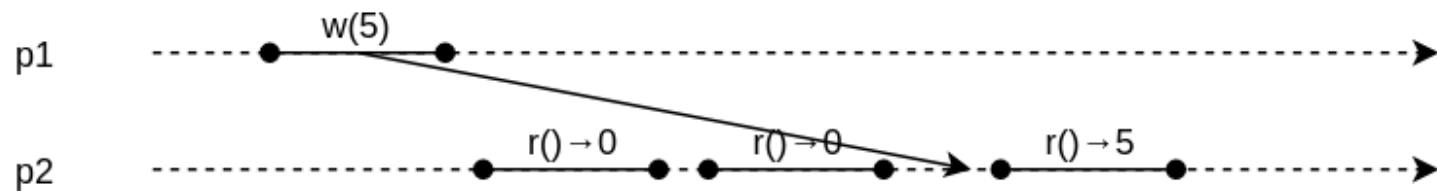
What is the issue with this algorithm?

## Decentralized algorithm (bogus)

- Intuitively, make an algorithm in which
  - A `read()` reads the local value.
  - A `write( $v$ )` writes to all nodes.
- To `read()`:
  - Return local value.
- To `write( $v$ )`:
  - Update local value to  $v$ .
  - Broadcast  $v$  to all (each node then locally updates).
  - Return.

### Exercise

What is the issue with this algorithm?



Validity is violated!



# Read-one Write-all algorithm

- Bogus algorithm modified.
- To `read()`:
  - Return local value.
- To `write( $v$ )`:
  - Update local value to  $v$ .
  - Broadcast  $v$  to all (each node locally updates).
  - Wait for acknowledgement from all correct nodes.
    - Require a perfect failure detector (fail-stop).
  - Return.

**Implements:**

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

**Uses:**

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

**upon event**  $\langle \textit{onrr}, \textit{Init} \rangle$  **do**

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

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

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

**upon event**  $\langle \textit{onrr}, \textit{Read} \rangle$  **do**

**trigger**  $\langle \textit{onrr}, \textit{ReadReturn} \mid \textit{val} \rangle$ ;

**upon event**  $\langle \textit{onrr}, \textit{Write} \mid v \rangle$  **do**

**trigger**  $\langle \textit{beb}, \textit{Broadcast} \mid [\text{WRITE}, v] \rangle$ ;

**upon event**  $\langle \textit{beb}, \textit{Deliver} \mid q, [\text{WRITE}, v] \rangle$  **do**

*val* := *v*;  
**trigger**  $\langle \textit{pl}, \textit{Send} \mid q, \text{ACK} \rangle$ ;

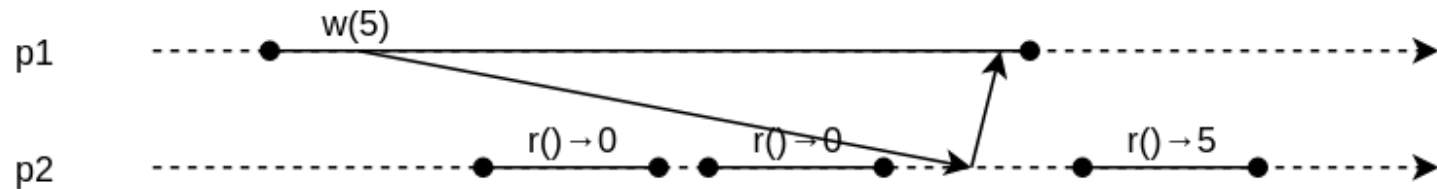
**upon event**  $\langle \textit{pl}, \textit{Deliver} \mid p, \text{ACK} \rangle$  **do**

*writeset* := *writeset*  $\cup \{p\}$ ;

**upon** *correct*  $\subseteq$  *writeset* **do**

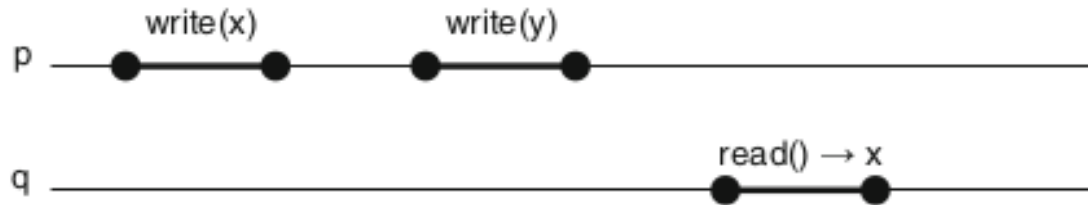
*writeset* :=  $\emptyset$ ;  
**trigger**  $\langle \textit{onrr}, \textit{WriteReturn} \rangle$ ;

## Read-one Write-all example



Validity is no longer violated because the write response has been postponed.

# Fail-silent algorithm



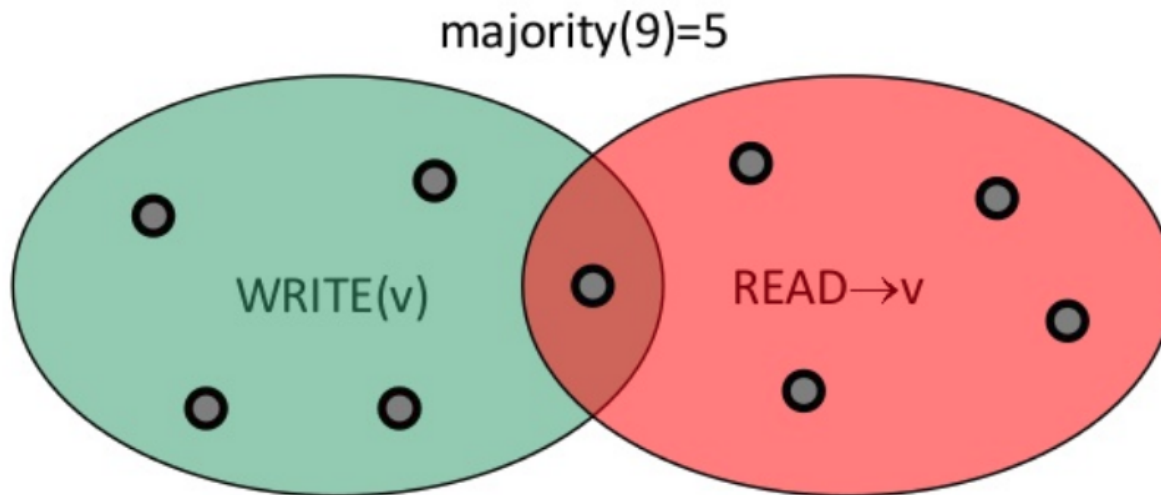
What if the failure detector is not perfect?

Can we implement a regular register in **fail-silent**? (without a failure detector)

- When writing, timestamp the value to be written.
- Always write to a strict majority of nodes.
- Always read from a strict majority of nodes, and pick the most recent value.

Then, provided that a strict majority of nodes are correct, validity is ensured.

## Quorum principle



# Majority voting algorithm

**Implements:**

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

**Uses:**

BestEffortBroadcast, **instance** *beb*;

PerfectPointToPointLinks, **instance** *pl*.

**upon event**  $\langle \textit{onrr}, \textit{Init} \rangle$  **do**

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

$wts := 0$ ;

$acks := 0$ ;

$rid := 0$ ;

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

**upon event**  $\langle \textit{onrr}, \textit{Write} \mid v \rangle$  **do**

$wts := wts + 1$ ;

$acks := 0$ ;

**trigger**  $\langle \textit{beb}, \textit{Broadcast} \mid [\textit{WRITE}, wts, v] \rangle$ ;

**upon event**  $\langle \textit{beb}, \textit{Deliver} \mid p, [\textit{WRITE}, ts', v'] \rangle$  **do**

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

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

**trigger**  $\langle \textit{pl}, \textit{Send} \mid p, [\textit{ACK}, ts'] \rangle$ ;

**upon event**  $\langle \textit{pl}, \textit{Deliver} \mid q, [\textit{ACK}, ts'] \rangle$  **such that**  $ts' = wts$  **do**

$acks := acks + 1$ ;

**if**  $acks > N/2$  **then**

$acks := 0$ ;

**trigger**  $\langle \textit{onrr}, \textit{WriteReturn} \rangle$ ;

```

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

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

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

```

## Exercise

Why do we reset `acks` and `readlist` right after having received back just more than  $N/2$  messages?



# Atomic registers

Towards single storage illusion.

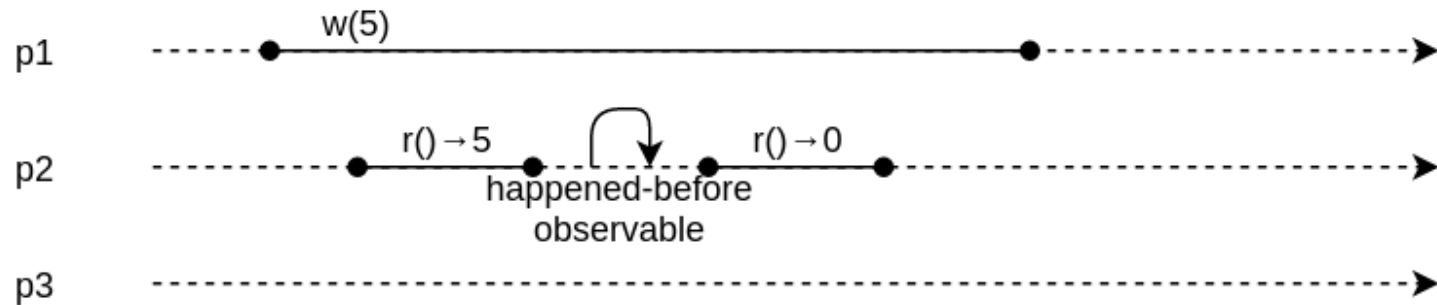
# Sequential consistency

An operation  $o_1$  locally precedes  $o_2$  in  $E$  if  $o_1$  and  $o_2$  occur at the same node and  $o_1$  precedes  $o_2$  in  $E$ .

An execution  $E$  is sequentially consistent if an execution  $F$  exists such that:

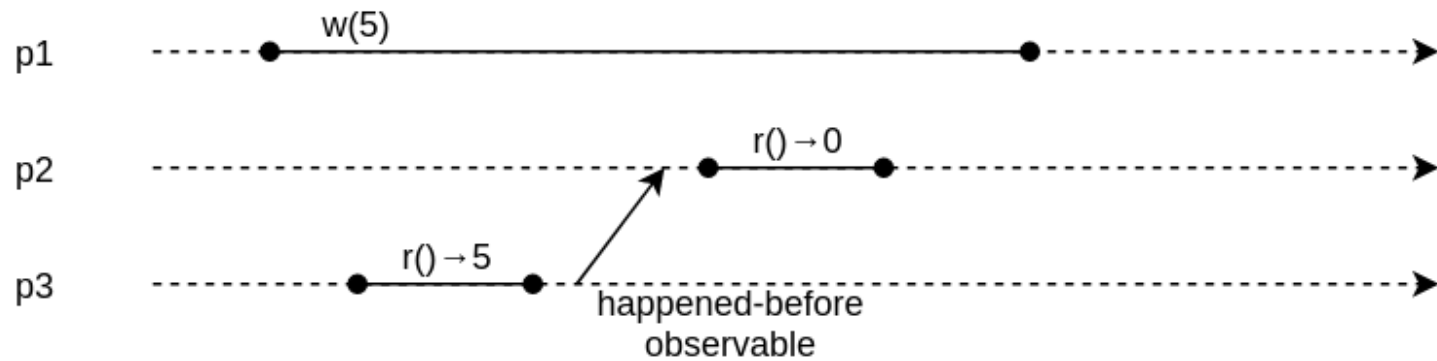
- $E$  and  $F$  contain the same events;
- $F$  is sequential;
- Read responses have value of the preceding write invocation in  $F$ ;
- If  $o_1$  locally precedes  $o_2$  in  $E$ , then  $o_1$  locally precedes  $o_2$  in  $F$ .

## Example (1)



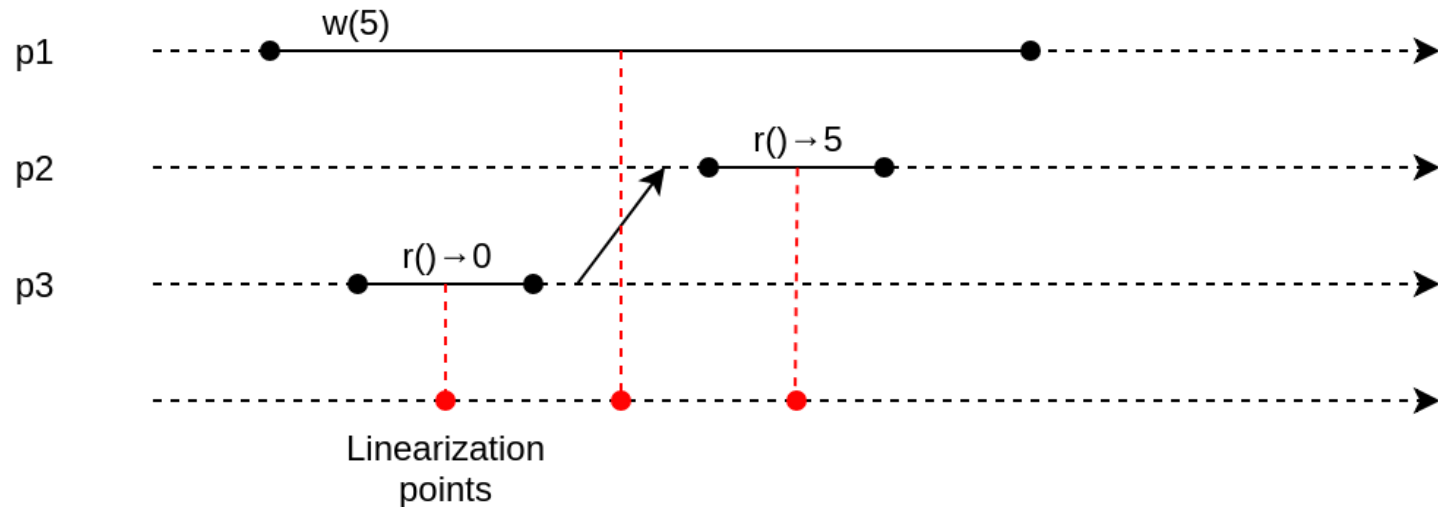
Sequential consistency **disallows** such execution.

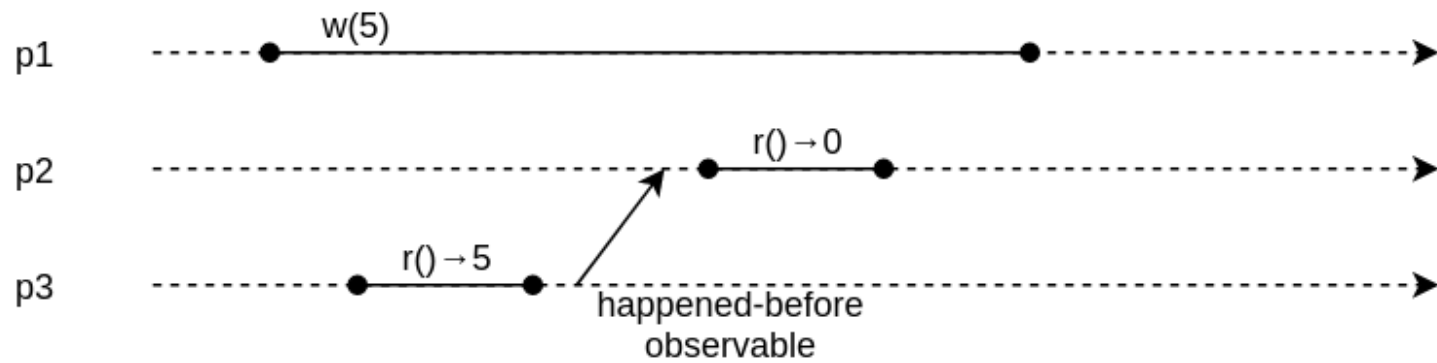
## Example (2)



Sequential consistency **allows** such execution.

# Linearization





This execution is sequentially consistent but is not linearizable!

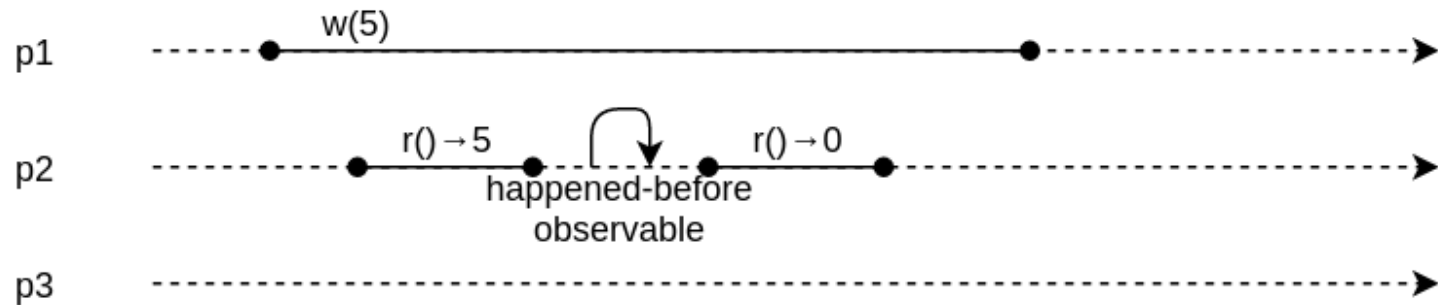
- **Linearizability:**

- Read operations appear as if **immediately** happened at all nodes at time between invocation and response.
- Write operations appear as if **immediately** happened at all nodes at time between invocation and response.
- Failed operations appear as
  - completed at every node, XOR
  - never occurred at any node.
- The hypothetical serial execution is called a **linearization** of the actual execution.

- **Termination:**

- If node is correct, each read and write operation eventually completes.

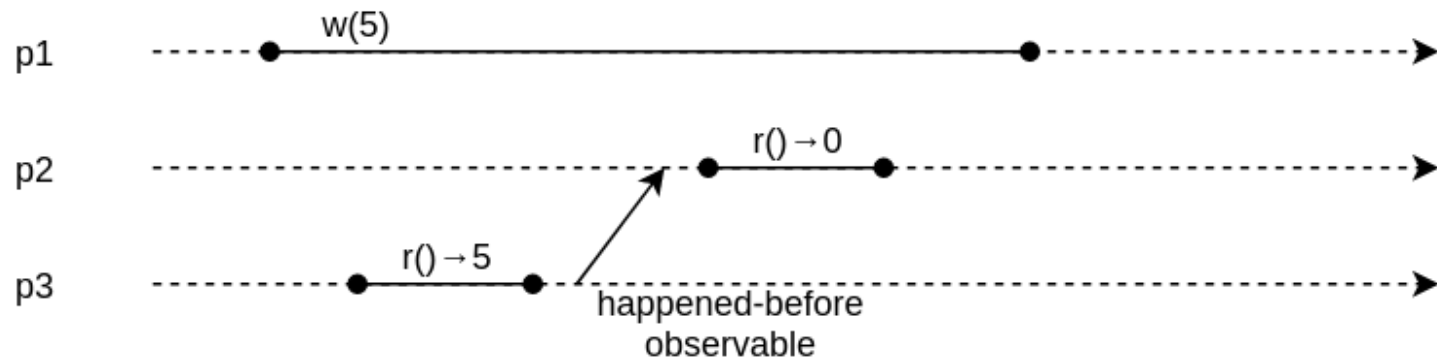
## Example (1)



Linearizability **disallows** such execution.



## Example (2)



Linearizability **disallows** such execution.

# $(1, N)$ atomic registers

## 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:

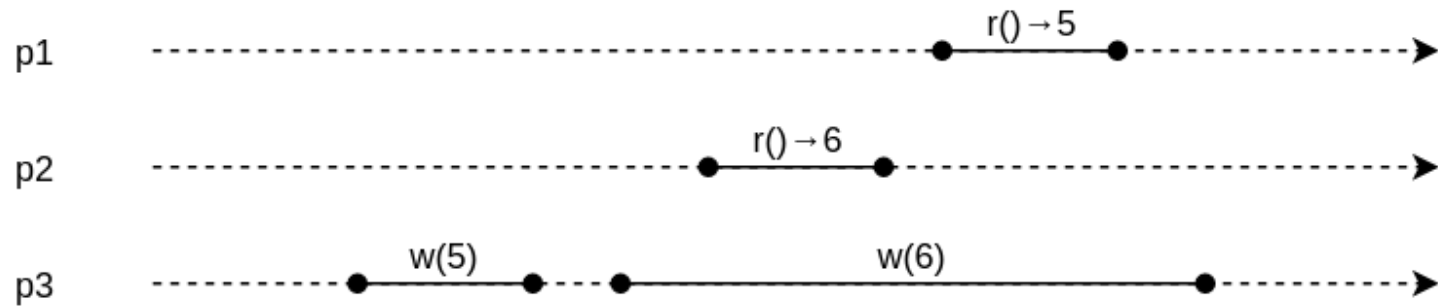
**ONAR1–ONAR2:** Same as properties ONRR1–ONRR2 of a  $(1, N)$  regular register (Module 4.1).

**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  $v$ .

## Exercise

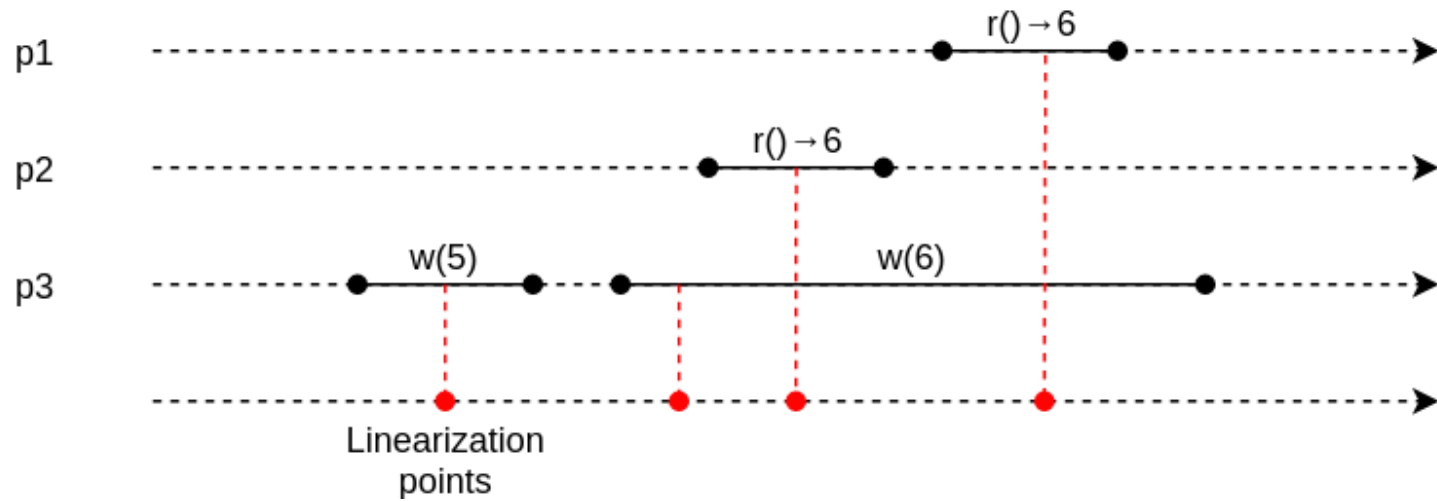
Show that linearizability is equivalent to validity + ordering.

## Atomic register example (1)



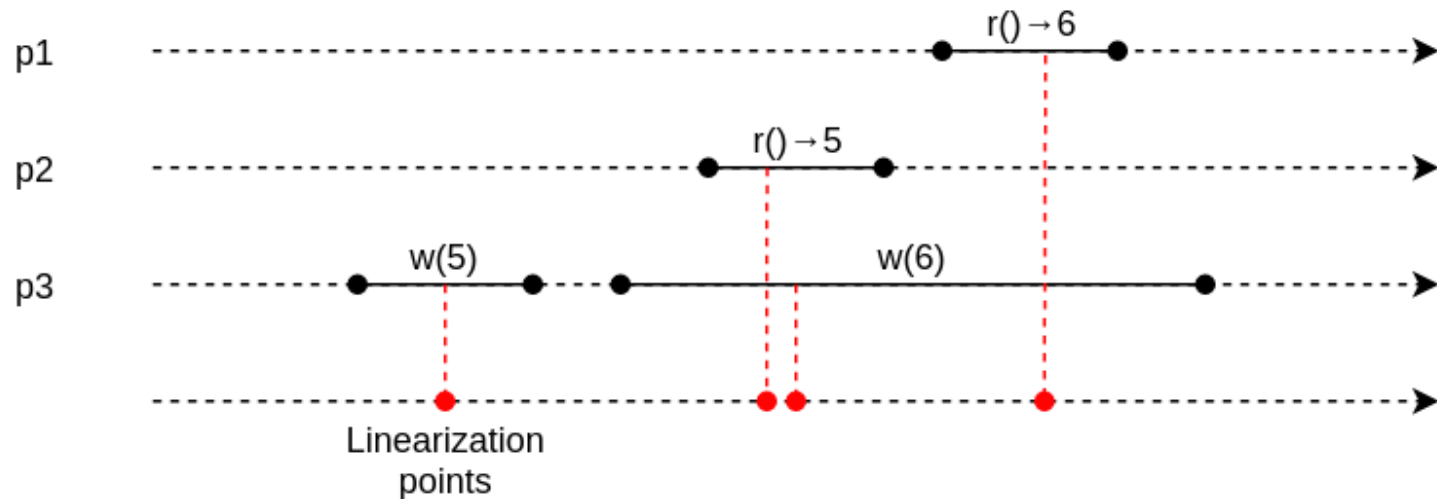
[Q] Atomic?

## Atomic register example (2)



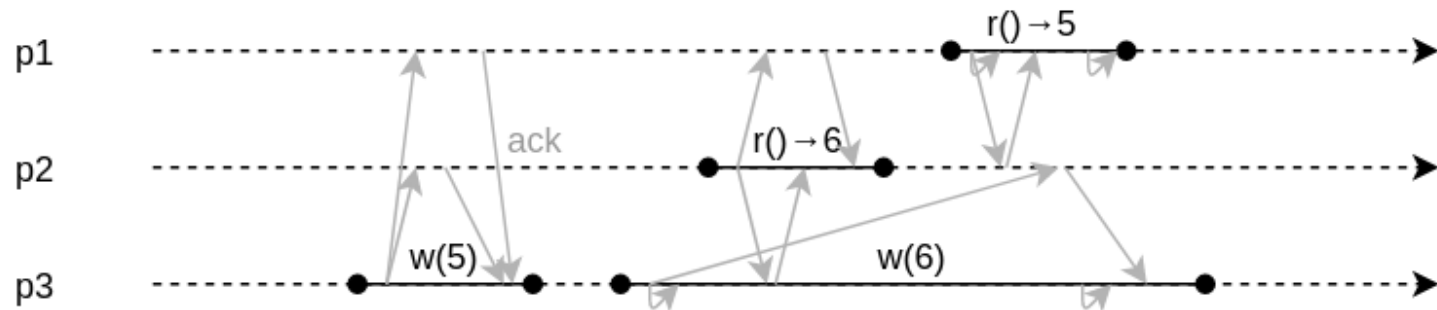
[Q] Atomic?

### Atomic register example (3)



[Q] Atomic?

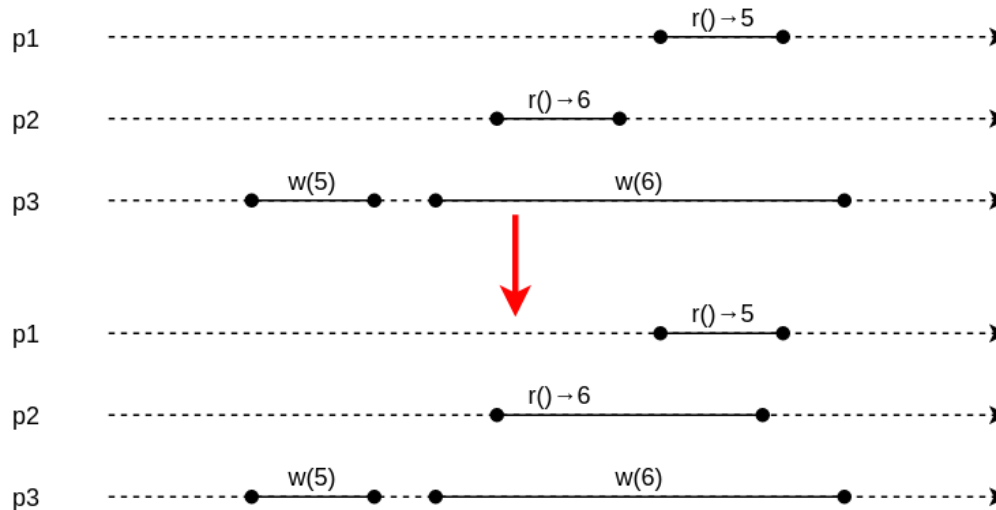
## Regular but not atomic



[Q] Atomic? Regular?

# Implementation of $(1, N)$ atomic registers

- When reading, write back the value that is about to be returned.
- Maintain a local timestamp  $ts$  and its associated value  $val$ .
- Overwrite the local pair only upon a write operation of a more recent value.



# Read-Impose Write-all algorithm

## Implements:

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

## Uses:

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

**upon event**  $\langle \textit{onar}, \textit{Init} \rangle$  **do**

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

$correct := \Pi$ ;

$writeset := \emptyset$ ;

$readval := \perp$ ;

$reading := \text{FALSE}$ ;

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

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

**upon event**  $\langle \textit{onar}, \textit{Read} \rangle$  **do**

$reading := \text{TRUE}$ ;

$readval := val$ ;

**trigger**  $\langle \textit{beb}, \textit{Broadcast} \mid [\text{WRITE}, ts, val] \rangle$ ;



```

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

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

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

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

```

## Exercise

How to adapt to fail-silent?

## Correctness

- **Ordering:** if a read returns  $v$  and a subsequent read returns  $w$ , then the write of  $w$  does not precede the write of  $v$ .
  - $p$  writes  $v$  with timestamp  $ts_v$ .
  - $p$  writes  $w$  with timestamp  $ts_w > ts_v$ .
  - $q$  reads the value  $w$ .
  - some time later,  $r$  invokes a read operation.
  - when  $q$  completes its read, all correct processes (including  $r$ ) have a timestamp  $ts \geq ts_w$ .
  - there is no way for  $r$  to change its value back to  $v$  after this because  $ts_v < ts_w$ .

### Exercise

Show that the termination and validity properties are satisfied.

# $(N, N)$ atomic registers

## Module:

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

## Events:

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

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

**Indication:**  $\langle nnar, ReadReturn \mid v \rangle$ : 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.

- How do we handle **multiple writers**?
- Read-Impose Write-all does not support multiple writers:
  - Assume  $p$  and  $q$  both store the same timestamp  $ts$  (e.g., because of a preceding completed operation).
  - When  $p$  and  $q$  proceed to write, different values would become associated with the same timestamp.
- Fix:
  - Together with the timestamp, pass and store the identity  $pid$  of the process that writes a value  $v$ .
  - Determine which message is the latest
    - by comparing timestamps,
    - by breaking ties using the process IDs.

# Simulating message passing?

- As we saw, we can simulate shared memory with message passing.
  - A majority of correct nodes is all that is needed.
- Can we **simulate message passing** in shared memory?
  - Yes: use one register  $pq$  for every channel.
    - Modeling a directed channel from  $p$  to  $q$ .
  - Send messages by appending to the right channel.
  - Receive messages by busy-polling incoming "channels".
- Shared memory and message passing are **equivalent**.

# Summary

- Shared memory registers form a **shared memory abstraction** with read and write operations.
  - Consistency of the data is guaranteed, even in the presence of failures and concurrency.
- Regular registers:
  - Bogus algorithm (does not work)
  - Centralized algorithm (if no failures)
  - Read-One Write-All algorithm (fail-stop)
  - Majority voting (fail-silent)
- Atomic registers:
  - Single writers
  - Multiple writers

