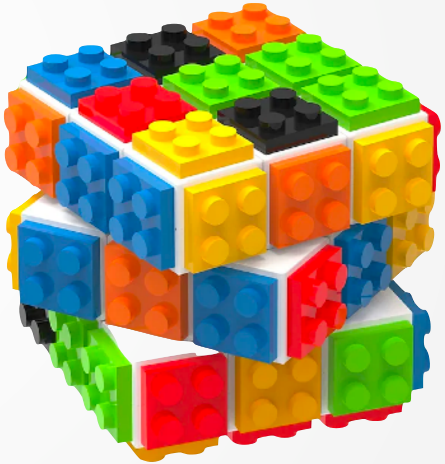
# Advanced Course
# Distributed Systems

## Distributed
## Shared Memory



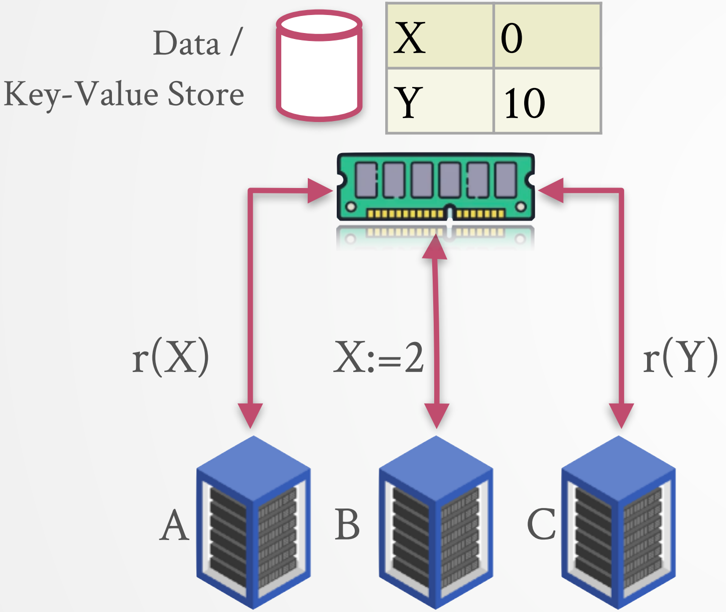
Paris Carbone

# Course Topics

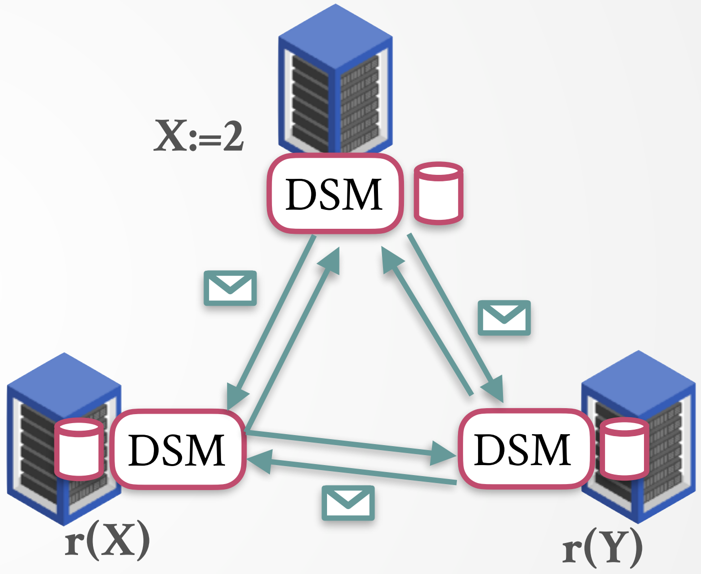
- Intro to Distributed Systems
- Fundamental Abstractions and Failure Detectors
- Reliable and Causal Order Broadcast
- Distributed Shared Memory/CRDTs
- Consensus (Paxos)
- Replicated State Machines (OmniPaxos, Raft, Zab etc.)
- Time Abstractions and Interval Clocks (Spanner etc.)
- Consistent Snapshotting (Stream Data Management)
- Distributed ACID Transactions (Cloud DBs)

# SHARED VS DISTRIBUTED SHARED "MEMORY"

## Registers / Key-Values




| | |
|---|---|
| X | 0 |
| Y | 10 |




**Shared Mem:** Processes/Servers have direct memory access (no messages)

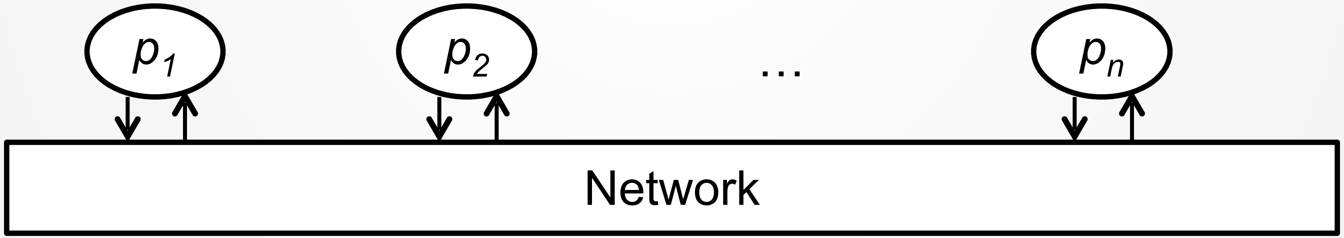
**Distributed Shared Mem (DSM):** Processes/Servers have indirect memory access (using messages)

# DISTRIBUTED SHARED MEMORY

- Provide *shared-memory as-a-service* (*simulated on message passing*).
  - Foundation of most **replicated** "key-value stores" today.
  - Algorithms suffice for simple read/write operations.
  - A ***register*** represents each memory location
    - Registers aka objects
  - Processes can ***read/write*** to a set of registers
    - More complex operations can be composed (FIFO-queue…)

# System Model

- Asynchronous system with *n* processes that communicate by message-passing
- Processes are automata with states and transitions as described by algorithm

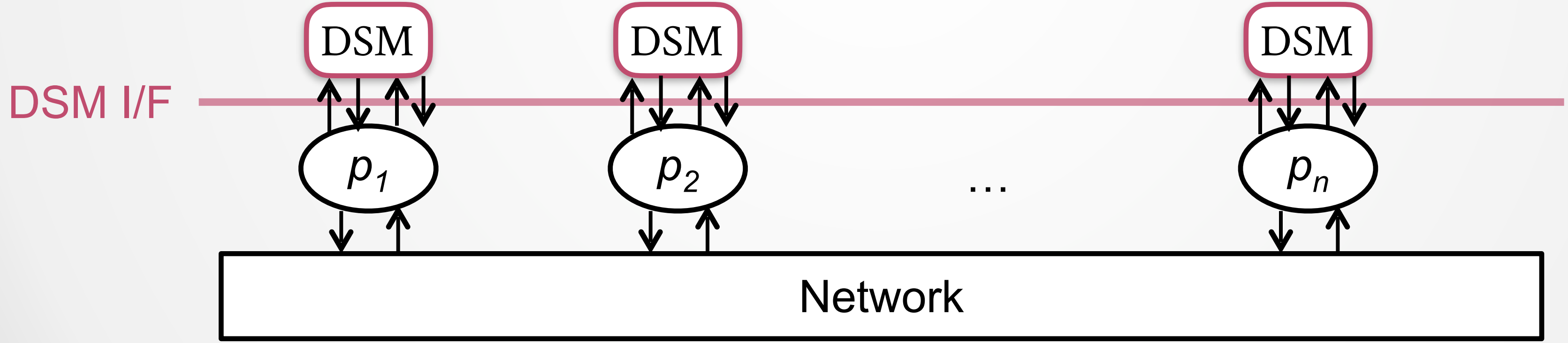


# READ/WRITE REGISTER

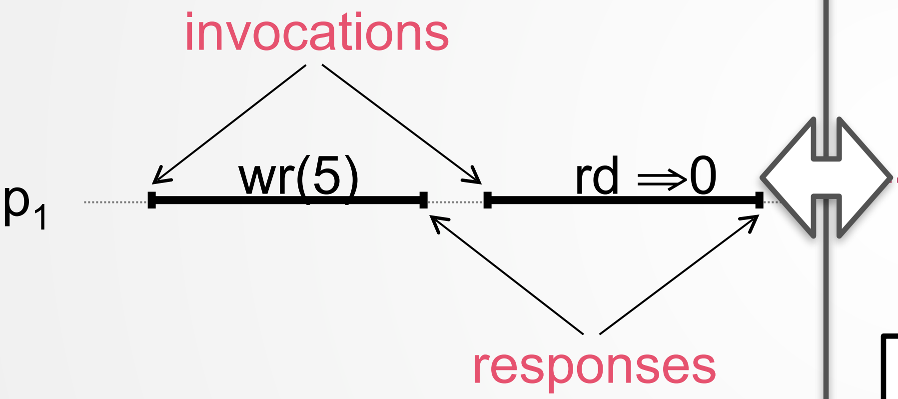
- RW-registers have 2 operations
  - read(r)⇒v
    - Value of $X_r$ was read to be v
  - write(r, v)
    - Update register $X_r$ to value v
  - Sometimes omit $X_r$
    - Specification with respect to one register

# Distributed Shared Memory

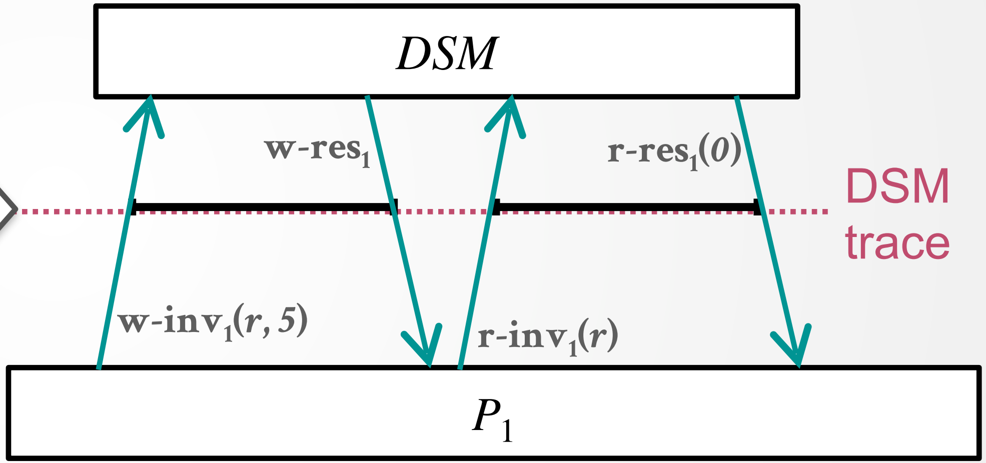
- DSM implements:
  - A set of read/write registers $\{x_r\}_{r \in \{1..m\}}$
  - Operations:
    - write($r$, $v$) – update value of register $x_r$ to $v$
    - read($r$) – return current value of register $x_r$



# 1 OPERATION = 2 EVENTS




Operations Defined by Invocations and Responses



# Basic Assumptions

- Processes are **sequential** (no pipelining of operations)
  - invocation, response, invocation, response,…
  - I.e. do one operation at a time
- Registers values of some type with some initial value of that type
  - Registers are of the integer type
  - Values are integers, initially zero

# TRACES (HISTORIES) OF EXECUTIONS

- Every trace consists of a sequences of events
  - r-inv$_i$(r)
    - Read invocation by process pi on register $X_r$
  - r-res$_i$(v)
    - Response with value v to read by process pi
  - w-inv$_i$(r,v)
    - Write invocation by process pi on register $X_r$ with value v
  - w-res$_i$
    - Response (confirmation) to write by process pi

# TRACE PROPERTIES

- Trace is **well-formed**
  - First event of every process is an invocation
  - Each process alternates between invocations and responses
- Trace is **sequential** if
  - $x$-inv by i immediately followed by a *corresponding* $x$-res at i
  - $x$-res by i immediately follows a *corresponding* $x$-inv by i
  - i.e. no concurrency, read x by p1, write y by p5, …
- Trace $T$ is **legal**
  - $T$ is sequential
  - Each read to $X_r$ returns last value written to register $X_r$

ID2203
KTH-2023

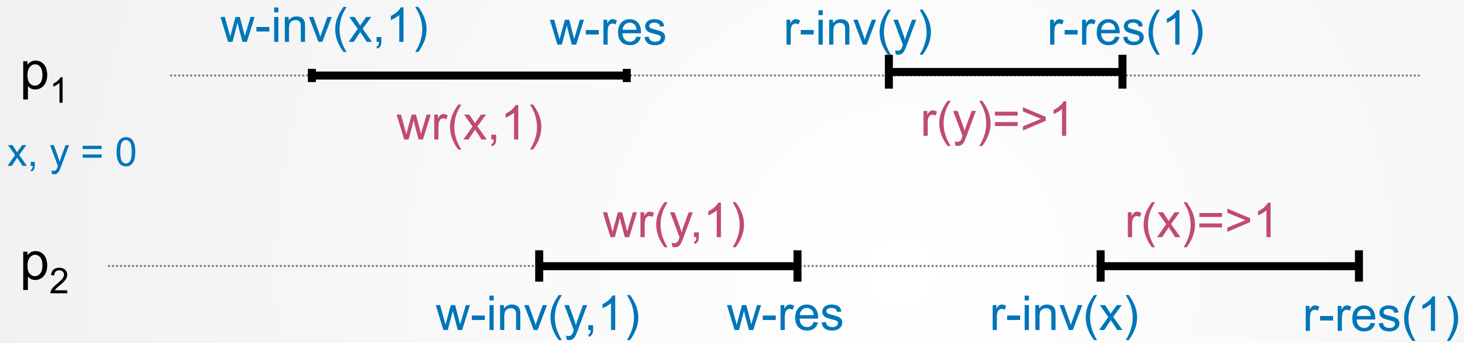
# Operation Properties

- An operation O of a trace T is
  - **complete** if both invocation & response occurred in T
  - **pending** if O invoked, but no response
- A trace *T is* **complete** if
  - Every operation is complete
  - Otherwise T is **partial**
- $op_1$ **precedes** $op_2$ in a trace T if (denoted $<_T$)
  - Response of $op_1$ precedes invocation of $op_2$ in T
- $op_1$ and $op_2$ are **concurrent** if neither precedes the other

ID2203

KTH-2023

# EXAMPLE




w-inv$_1$(x,1) w-inv$_2$(y,1) w-res$_1$ w-res$_2$ r-inv$_1$(y) r-inv$_2$(x) r-res$_1$(1) r-res$_2$(1)

# Regular Register Algorithms

# TERMINOLOGY

- (1,N)-algorithm
  - 1 designated writer, multiple readers

- (M,N)-algorithm
  - Multiple writers, multiple readers

# Regular Register (1, N)

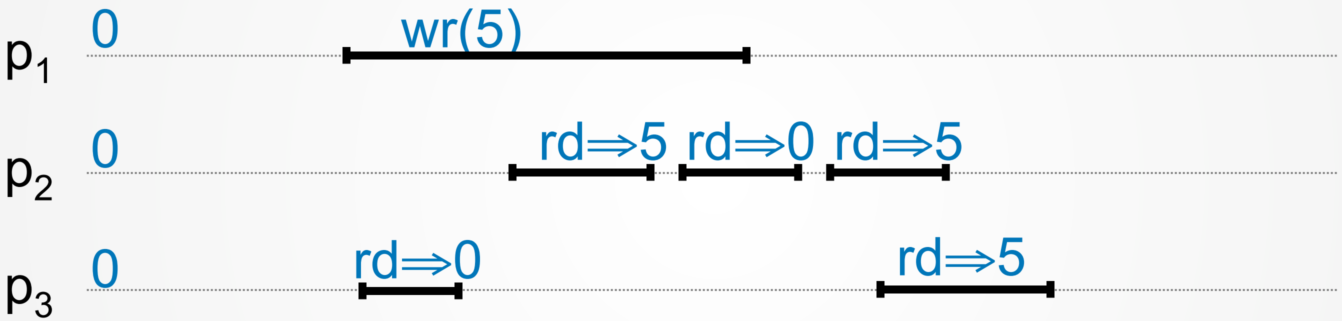
## Termination

- Each read/write operation issued by a correct process eventually completes.

## Validity

- Read returns ***last value written*** if
  - Not ***concurrent*** with another write, and
  - Not concurrent with a ***failed write***
- Otherwise may return last or concurrent "value"

# EXAMPLE




Regular? yes

Not a single storage illusion!

# CENTRALIZED ALGORITHM

Designate one process as ***leader***

- to **read**
  - Ask leader for latest value
- to **write(v)**
  - Update leader's value to v

- ***Problem?***
  - Does not work if leader crashes

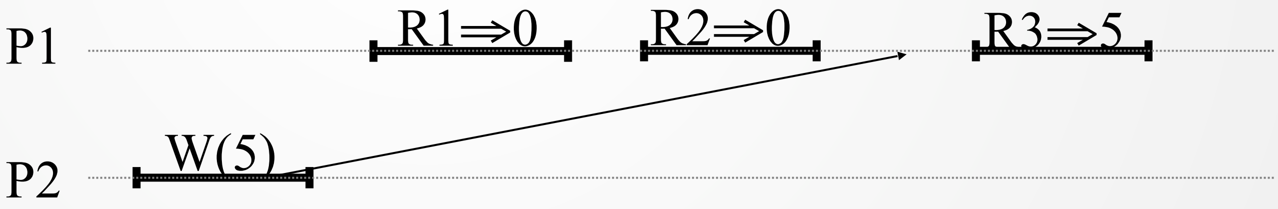
# STRAWMAN REGULAR ALGORITHM

- Intuitively: make an algorithm in which
  - A read just reads local value
  - A write writes to all processes

- to **write(v)**
  - Update local value to v
  - Broadcast v to all (each node locally updates)
  - Return

- to **read**
  - Return local value

- ***Problem?***



# FAIL-STOP READ-ONE WRITE-ALL (1,N)

- Bogus algorithm modified
  - Use perfect FD **P**
  - Fail-stop model
- to **write(v)**
  - Update local value to v
  - Broadcast v to all
  - ⏳ Wait for ACK from all ***correct processes***
  - Return
- to **read**
  - Return local value

# CORRECTNESS

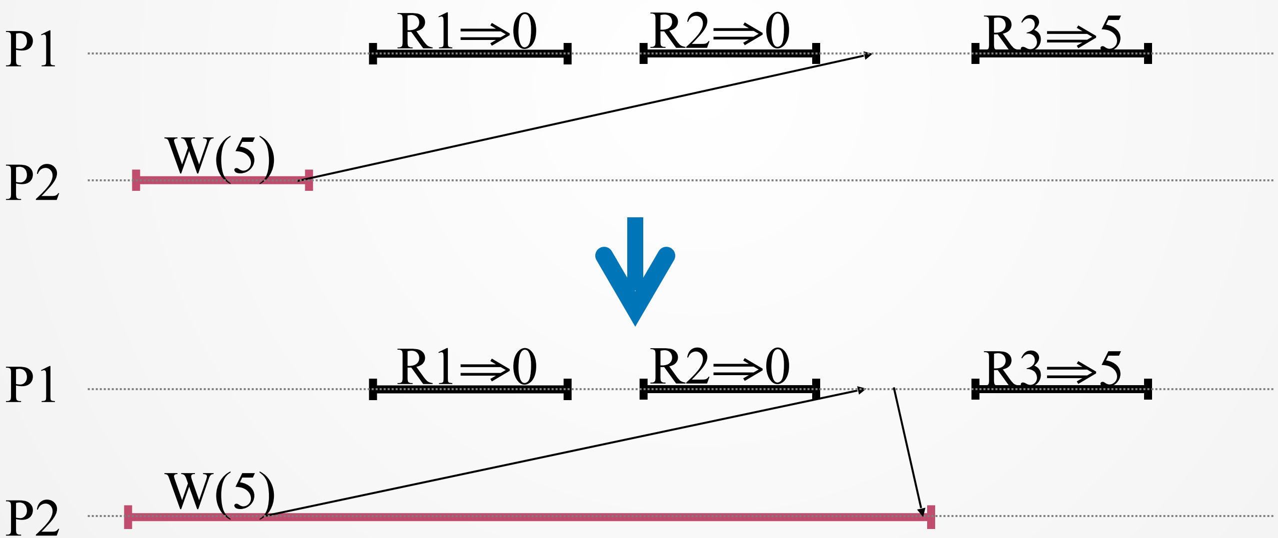
Assume we use Beb-broadcast, Perfect links and **P**

**Validity**

1. No concurrent write with the read operations
    - Assume p invokes a read, and v last written value
    - At time of read by p, the write is complete (accuracy of **P**) and p has v stored locally
2. Read is concurrent with write of value v, v' the value prior to v
    - Each process store v' before write(v) is invoked
    - When a read is invoked each process either stores v or v'
    - As the write is concurrent, **either value is correct to read**

# READ-ONE WRITE-ALL (1,N) #2

Intuitively Postpone write responses

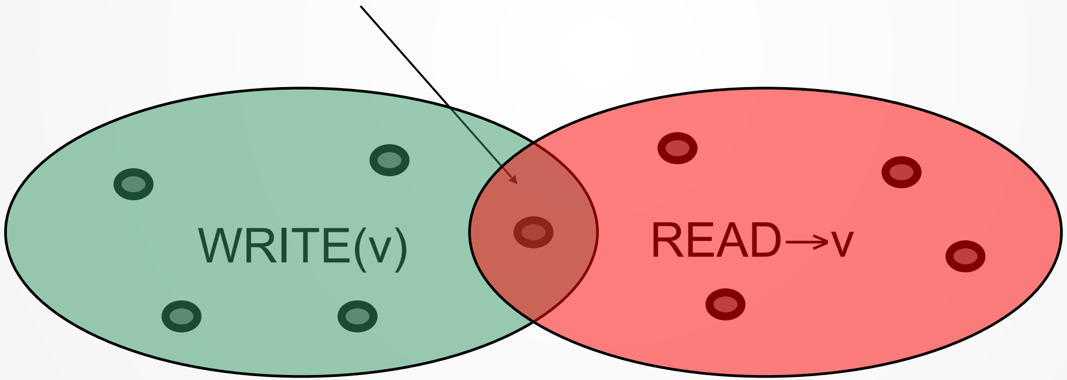


# Supporting Weaker Models

### *Main idea*

Quorum principle (ex: majority)

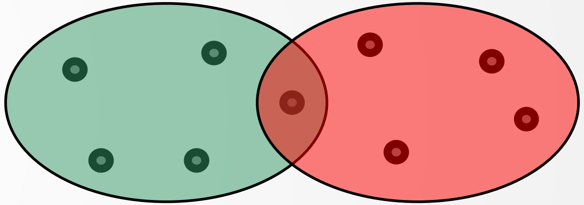
- Always write to and read from a majority of processes
- At least one correct process knows most recent value




Ex: majority(9)=5

# Quorum Principle

- Divide the system into quorums
  - Any two quorums should intersect (overlap)
  - E.g., read R, write W, s.t. $R+W>N$



- Majority Quorum
  - **Pro**: tolerate up to $\lceil N/2 \rceil -1$ crashes
  - **Con**: Have to read/write $\lfloor N/2 \rfloor +1$ values

# Timestamp-value pairs

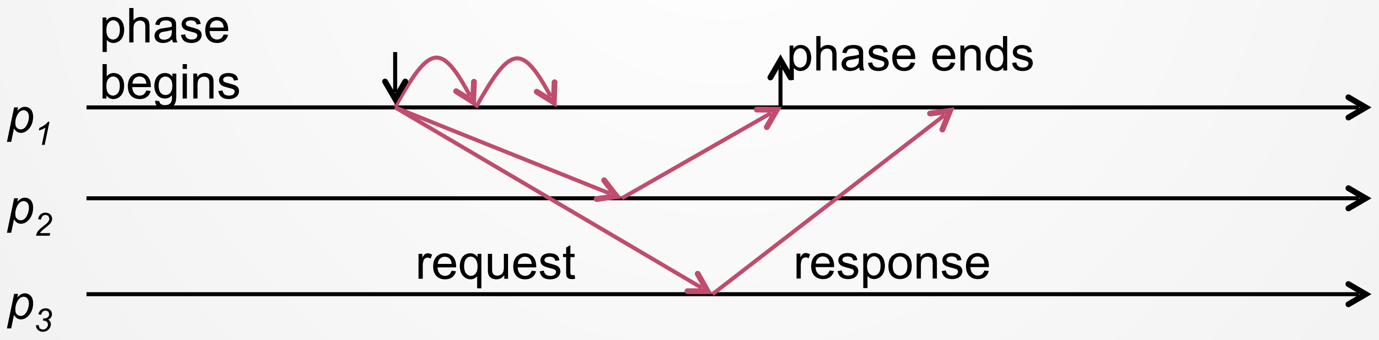
- Each process stores the values of all registers
- Value of register r
  - is timestamp-value pair, $tvp=(ts, v)$
  - *ts* is a sequence number initialized to zero at the writer and incremented at each write
  - *ts* determine which value is more recent
  - Initially r is (ts, val) = $(0, \perp)$ at all processes
- Each process
  - Stores the value of register r with max timestamp for each register r

# PHASES

Each operation is executed into *phases*

A phase run by $p_i$ consists of:

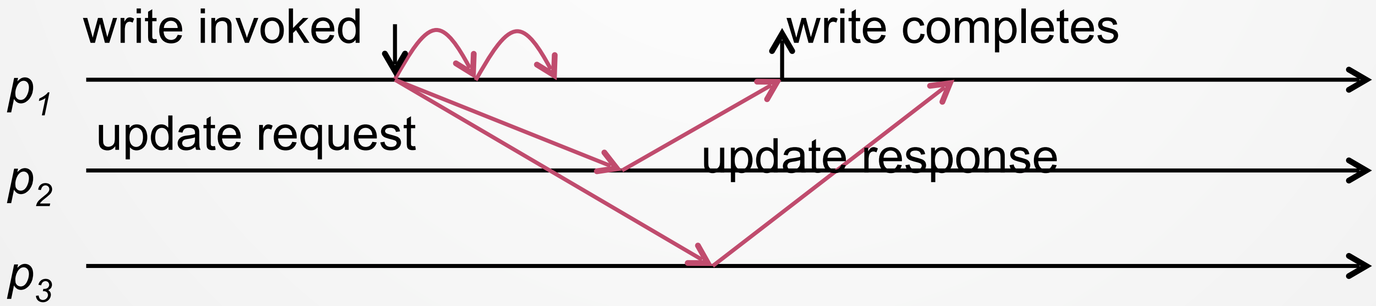
- $p_i$ beb-broadcasts a request
- $p_j$ receives request, processes it, and sends response
- $p_i$ waits for responses from a majority before the phase ends




ID2203
KTH-2023

# Write Majority

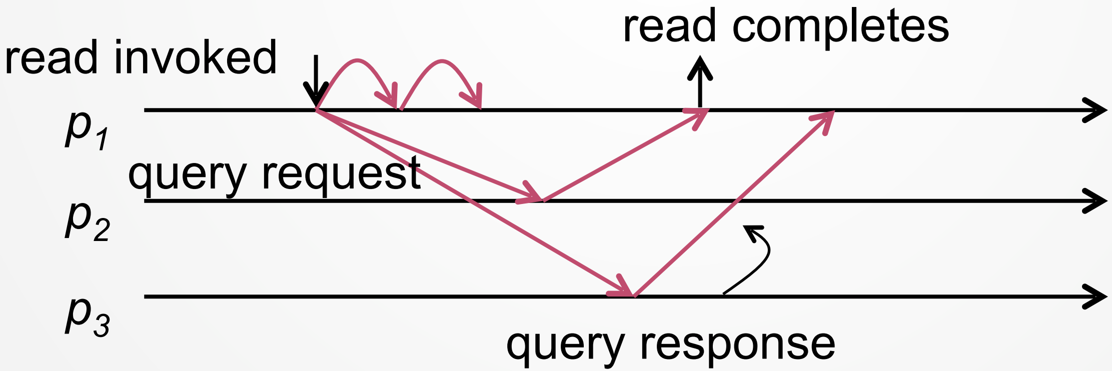
- Writer executing write($r$, $v$) operation
  - *ts++* (increment current sequence number)
  - $p_i$ forms *tvp*=(ts, $v$), where *ts* is current sequence number
  - $p_i$ starts an ***update phase*** by sending **update request** with register id $r$ and timestamp-value pair (ts, v)
  - $p_j$ updates r = **max(r, (ts, v))** and responds with ACK
  - $p_i$ completes write when update phase ends



# READ MAJORITY

Process $p_i$ executing read($r$) operation

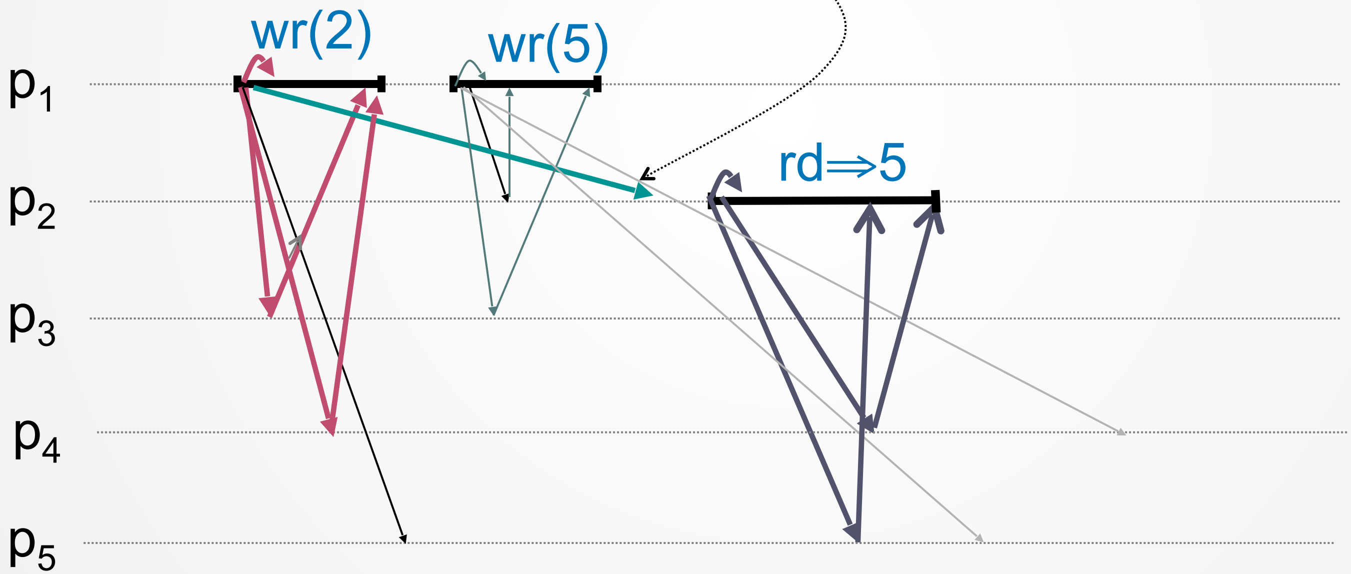
- $p_i$ starts **query phase**, sends query request with id $r$
- $p_j$ responds to the query with $(\mathrm{ts}, \mathrm{v})_j$
- When query phase ends, $p_i$ picks **max** $\mathbf{(ts, v)_j}$ **received**



# ILLUSTRATING MAJORITY VOTING ALGORITHM

Avoiding old writes overwriting new write

$p_j$ updates r $=$ **max(r, (ts, v))** and responds with ACK



# Correctness Validity

- No concurrent write with the read operations
  - Assume q invokes a read, and (ts,v) last written value by p. ts is highest time stamp.
  - At time of read-inv by q, a majority has (ts,v)
  - q gets at least one response with (ts,v) and returns v
- Read is concurrent with a write with value (ts,v)
  - (ts-1,v') the value prior to (ts,v)
  - Majority of processes store (ts-1,v') before write(v) is invoked
  - The query phase of the read returns either (ts-1,v') or (ts,v)

# Performance and Resilience

- **Read-one write-all (1,N) algorithm**
  - Time complexity (write)
    - 2 communication steps (broadcast and Ack)
  - Message complexity: O(N) messages
  - Resilience: faulty processes f = N-1
- **Majority voting (1,N) algorithm**
  - Time complexity (write and read)
    - 2 communication steps (one round trip)
  - Message complexity: O(N) messages
  - Resilience: faulty processes $f < \lceil N/2 \rceil$

# Towards single storage illusion...

# Atomic/Linearizability vs. Sequential Consistency

# Sequential Consistency

"the result of any execution is the same as if the operations of all the processes were executed in some sequential order, and the operations of each individual process in this sequence are in the order specified by its program"

# Linearizability/Atomic Consistency

"the result of any execution is the same as if the operations of all the processes were executed in some sequential order, and the operations in this sequence are in the global time order of operations (occurs bet. invocation and response)"

# SAFETY: CONSISTENCY INFORMALLY

- Sequential Consistency: only allow executions whose results appear as if there is a single system image and "local time" is obeyed.

- Linearizability/Atomicity: only allow executions whose results appear as if there is a single system image and "global time" is obeyed.

# Sequential Consistency Formally (SC)

- Trace S is **legal**
  - S is **sequential**
  - Each read to Xr returns last value written to register Xr
- Given a trace **T**, **T** $| p_i$ (view of process pi)
  - Subsequence of T with only $x\text{-}inv_i$ and $x\text{-}res_i$ of $p_i$
  - Traces **S** and **T** are **equivalent** (written as **S** $\simeq$ **T** )
  - if $\forall p_i$: $S|p_i = T|p_i$
- SC(T) as property on traces T:
  - SC(T) if **there exists legal history S** such that S$\simeq$T

# LINEARIZABILITY (LIN) FORMALLY

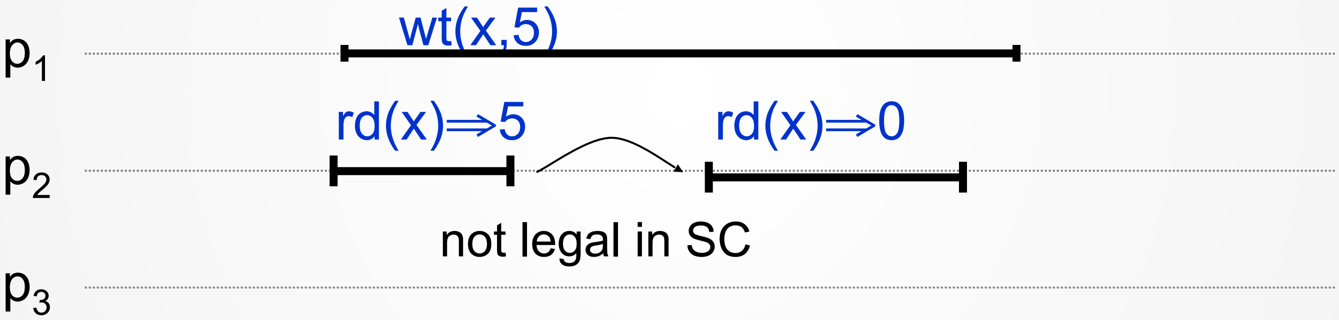
- LIN is a consistency condition similar to SC
  - LIN($T$) requires that there exists legal Trace $S$:
    - $S$ is equivalent to $T$,
    - **If $o_1 <_T o_2$ then it must also be that $o_1 <_S o_2$**
- LIN is stronger than SC: LIN($T$) $\Rightarrow$ SC($T$)

# CONSIDERING FAILURES

- No observable failures in complete executions
- **Linearizability** (or **SC**) for **partial executions** (failures)
  - A partial trace T is **linearizable** (or SC) if T is modified to T' s.t.
    - Every pending operation is **completed** by
      - **Removing** the **invocation** of the operation, or
      - **Adding response** to the operation
    - T' is linearizable (SC)

# SC EXAMPLE 1

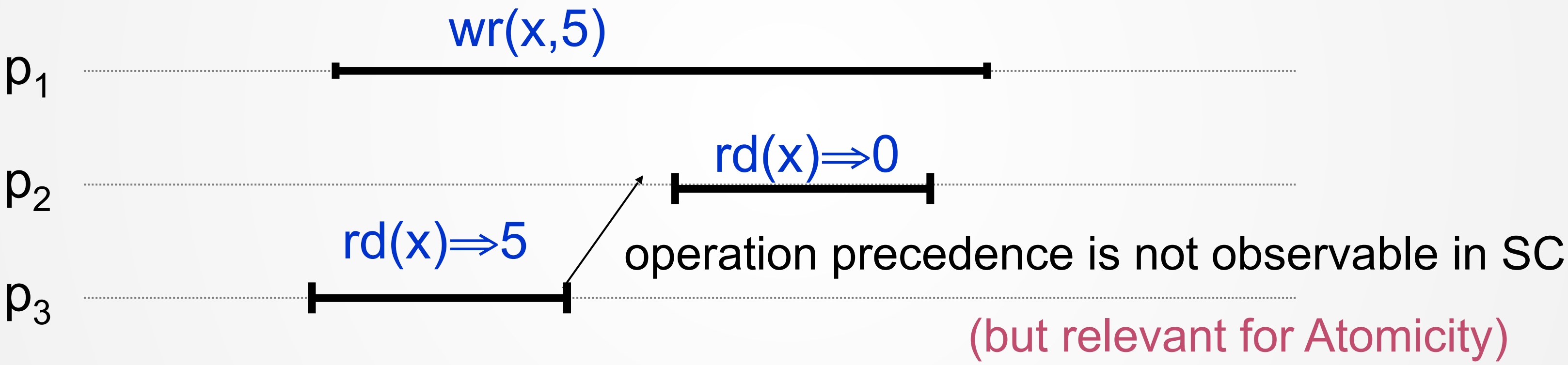
## Regular execution




Sequential consistency disallows such E's

# SC EXAMPLE 2

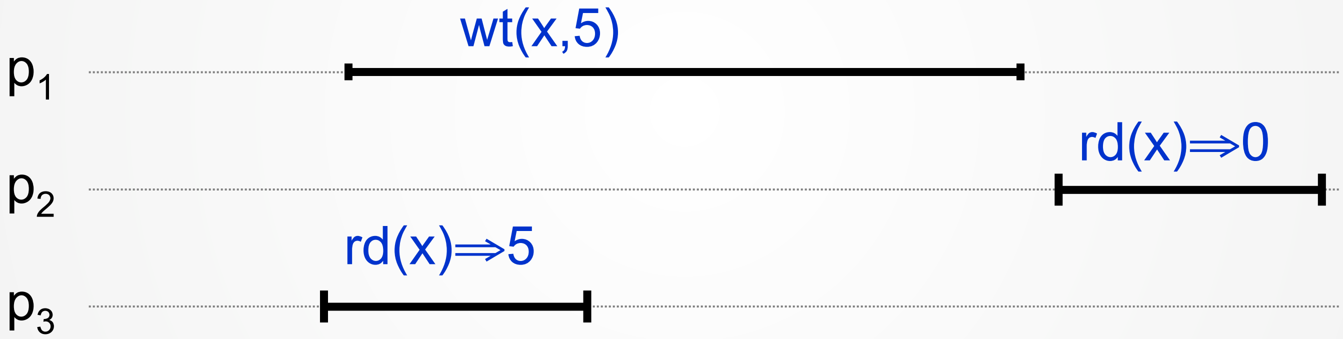
Regular execution




Sequential consistency allows such T's

# Regularity vs SC

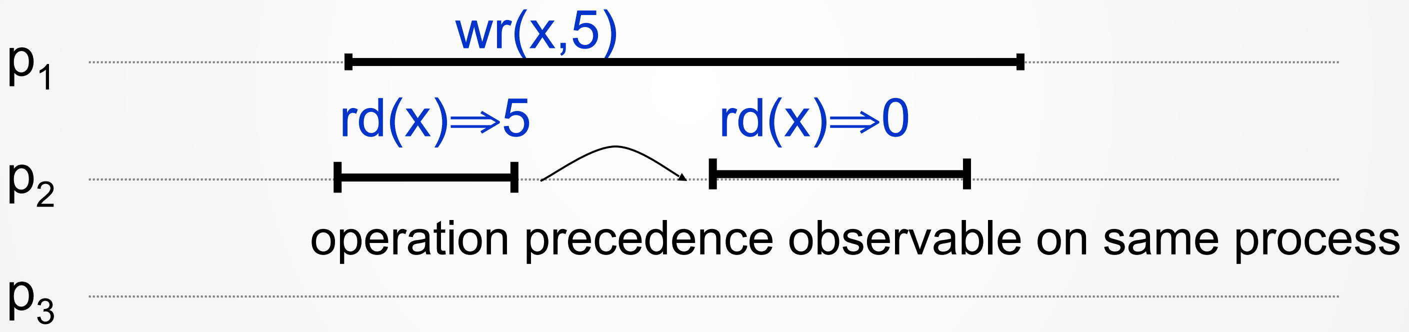
Sequentially consistent execution




Regular consistency disallows such trace

# ATOMICITY EXAMPLE 1

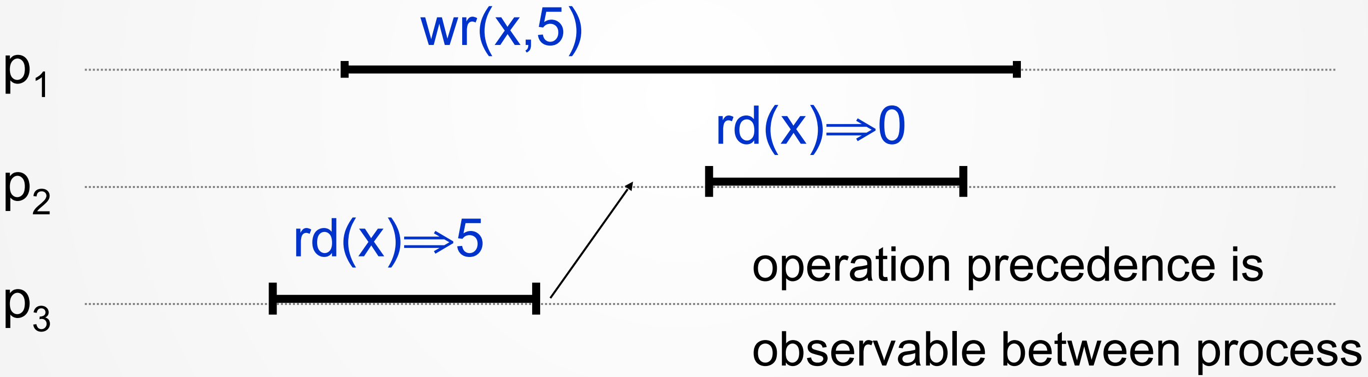
- Regular execution




- Atomicity/Linearizability disallows such E's
  - No single storage could behave that way

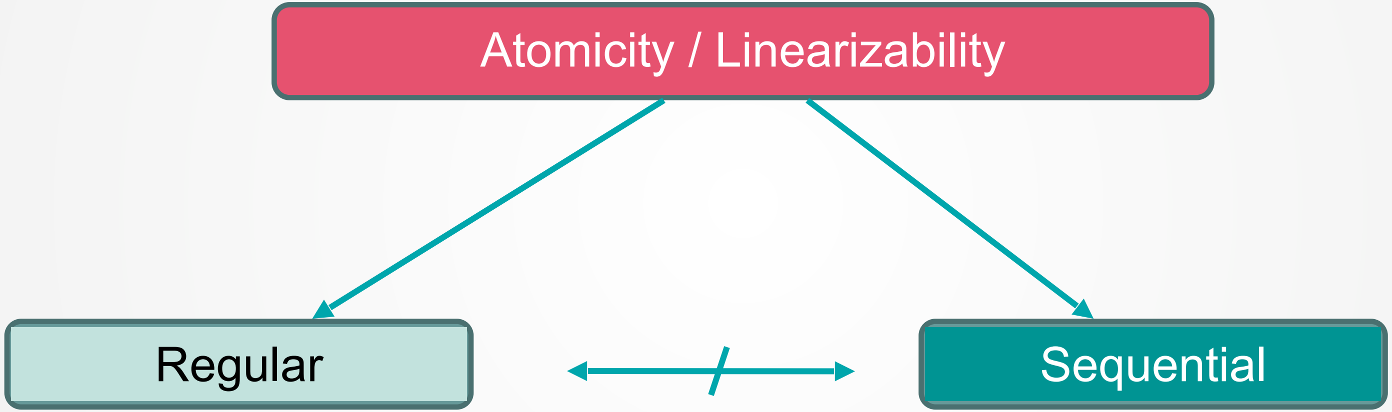
# Atomicity Example 2

- Regular execution




- Atomicity/Linearizability disallows such E's

# CONSISTENCY HIERARCHY



# COMPOSITIONALITY

- For a trace T
  - $T \mid x_r$ Subsequence of T with only $x$-inv and $x$-res of **register $\mathbf{x_r}$**
- For multi-registers, we would like to have modular design and verification of the algorithm that implements certain consistency model
- This is possible if we can design the algorithm for each register in isolation
- Possible with compositional consistency condition
  - Consistency condition CC(T) is compositional (local) iff
    - for all registers $x_r$: $CC(T \mid x_r)) \Leftrightarrow CC(T)$

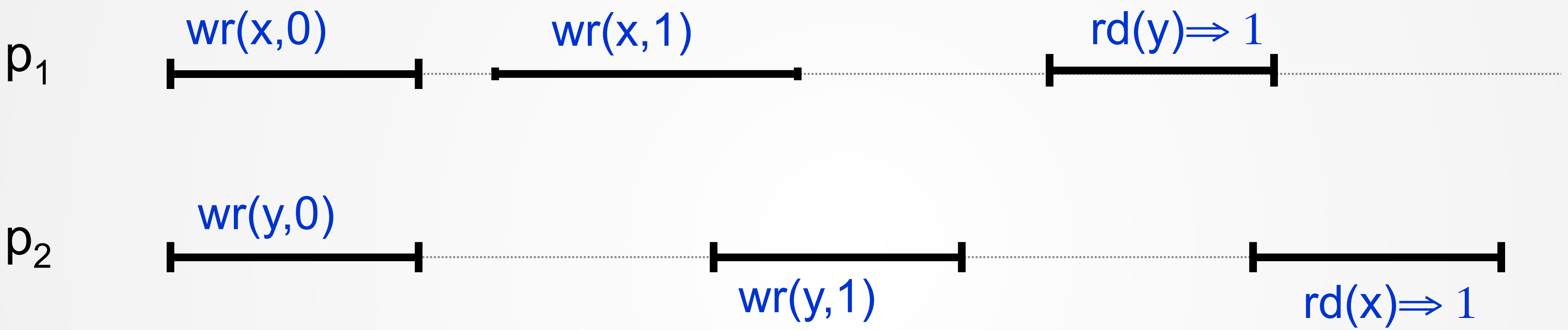
ID2203

KTH-2023

# Compositionality

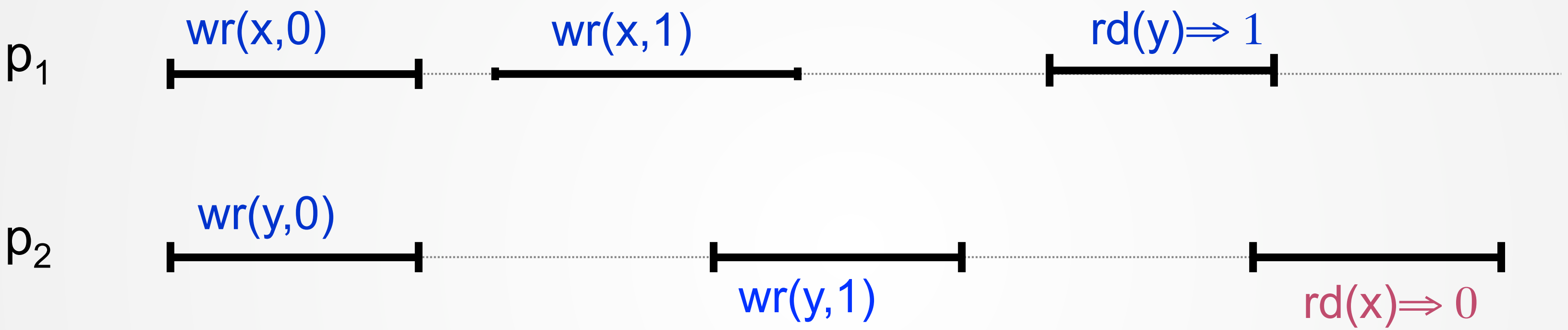
- Possible with compositional consistency condition
  - Consistency condition CC($H$) is compositional iff
    - $(\forall x_r\text{: } CC(H|x_r)) \Leftrightarrow CC(H)$
- Linearizability is compositional
  - for all registers $x_r$: LIN(T$|x_r$) $\Leftrightarrow$ LIN(T)
- Unfortunately, SC is not compositional
  - Even though we can show SC(T$|x_r$) for each register, SC(T) may not hold

# Example Linearizable Trace




$T$ : wr(x,0) wr(y,0) wr(y,1) wr(x,1) rd(y)⇒ 1 rd(x)⇒ 1

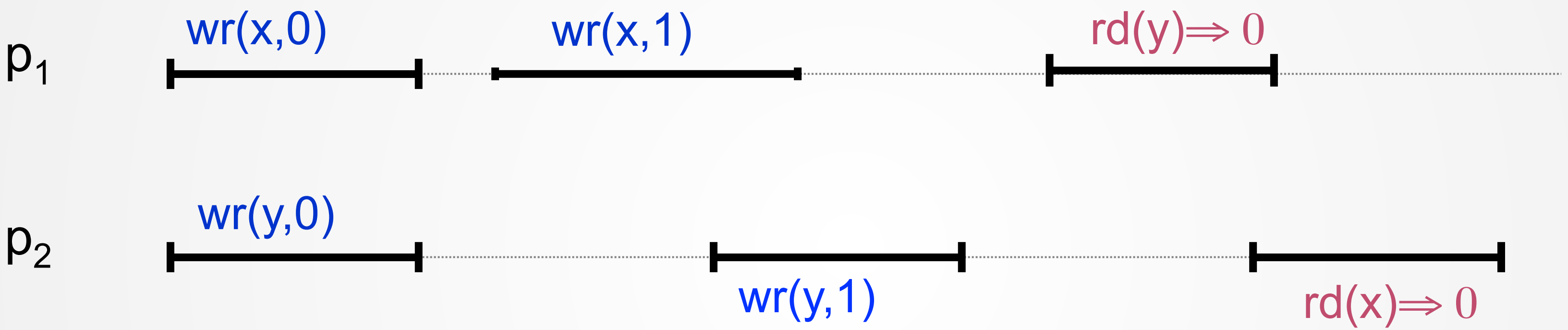
# Example Sequentially Consistent Trace

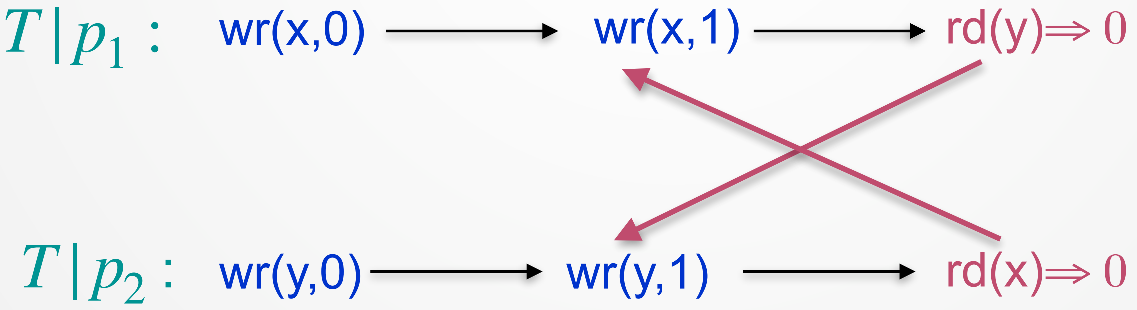



Legal History

$H$ : wr(x,0) wr(y,0) wr(y,1) rd(x)$\Rightarrow$ 0 wr(x,1) rd(y)$\Rightarrow$ 1

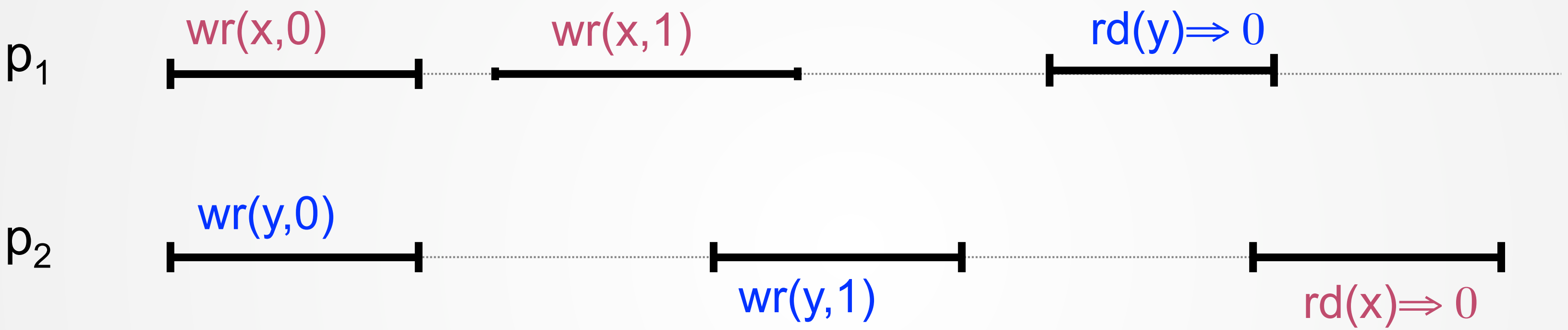
# NOT SEQUENTIALLY CONSISTENT TRACE




No legal history is possible

# Sequential Consistent is not Compositional




$$T|x: \; wr_1(\text{x,0}) \longrightarrow rd_2(\text{x}) \Rightarrow 0 \longrightarrow wr_1(\text{x,1})$$

$$T|y: \; wr_2(\text{y,0}) \longrightarrow rd_1(\text{y}) \Rightarrow 0 \longrightarrow wr_2(\text{y,1})$$

# LIVENESS: PROGRESS

- Liveness requirements
  - Wait-free
    - Informally:
      - Every correct node should "make progress"
      - (no deadlocks, no live-locks, no starvation)
  - Lock-free/non-blocking
    - Informally:
      - At least one correct node should "make progress"
      - (no deadlocks, no live-locks, maybe starvation)
  - Obstruction free/solo-termination
    - Informally:
      - if a single node executes without interference (contention) it makes progress
      - (no deadlocks, maybe live-locks, maybe starvation)

# Atomic/Linearizable Registers Algorithms

# Atomic/Linearizable Register

- Termination (Wait-freedom)
  - If node is correct, each read and write op eventually completes
- Linearization Points
  - **Read ops** appear as if **immediately** happened at all nodes at
    - time between invocation and response
  - **Write ops** appear as if **immediately** happened at all nodes at
    - time between invocation and response
  - **Failed ops** appear as
    - completed at every node, XOR
    - never occurred at any node

# ALTERNATIVE DEFINITION

## Linearization points

**Read ops** appear as **immediately** happened at all nodes at

time between invocation and response

**Write ops** appear as **immediately** happened at all nodes at

time between invocation and response

**Failed ops** appear as

completed at every node, XOR

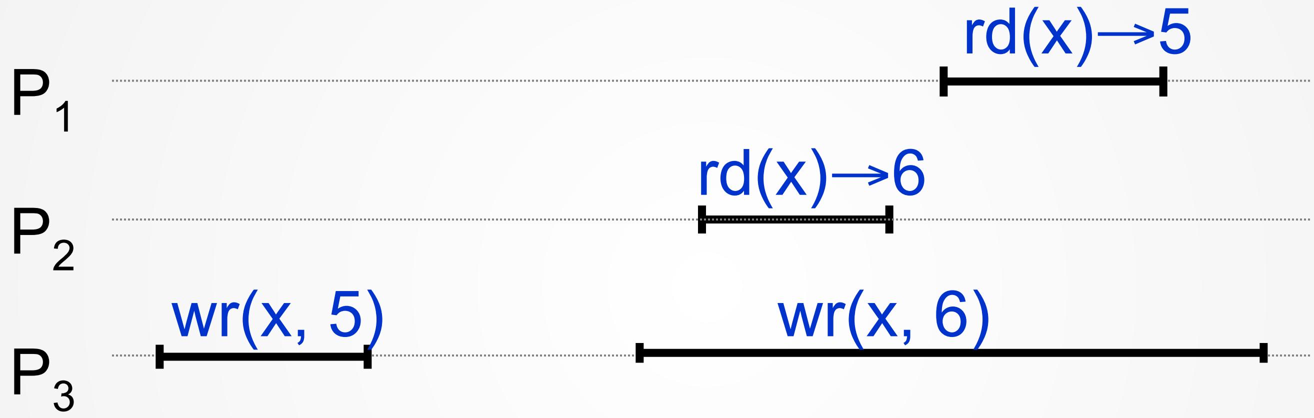
never happened at any node

=

## Ordering (only (1,N))

- **Validity**
  - Read returns last value written if
    - Not concurrent with another write
    - Not concurrent with a failed operation
  - Otherwise may return last or concurrent "value"
- **Ordering**
  - If read→r1 precedes read→r2 then write(r1) precedes write(r2)

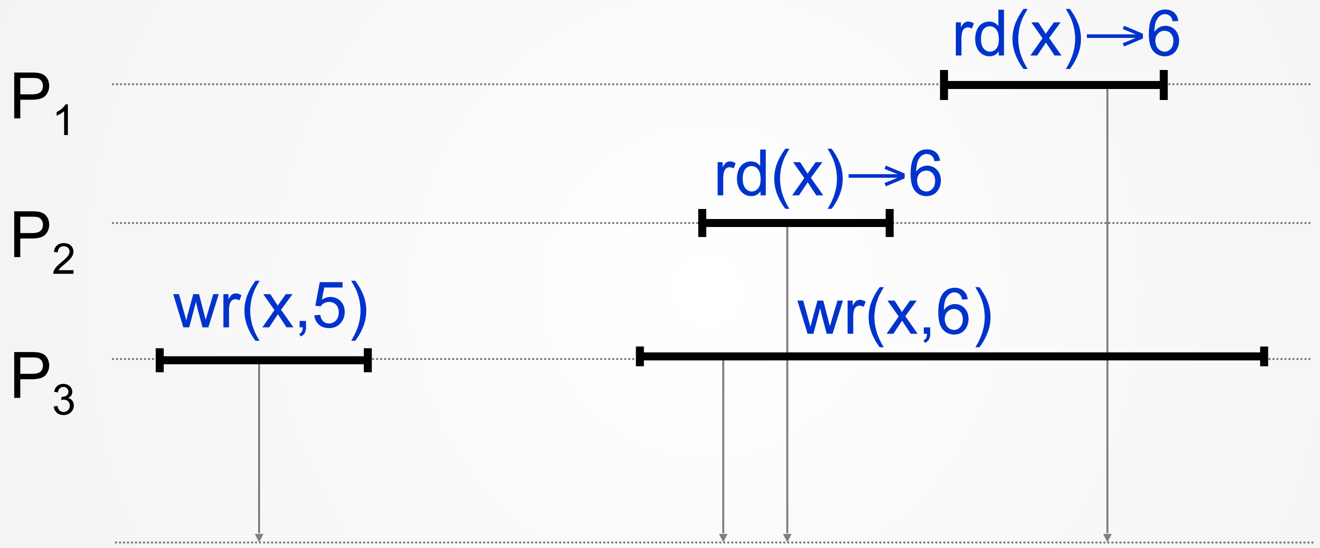
# EXAMPLE




Atomic?

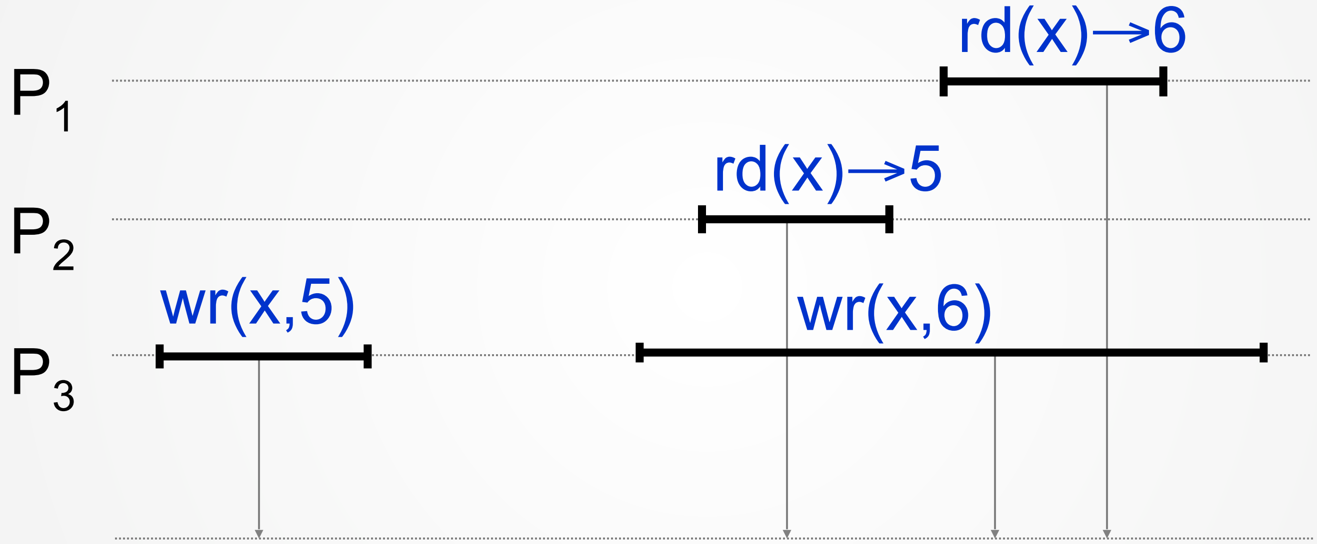
No, not possible to find linearization points

# Example 2



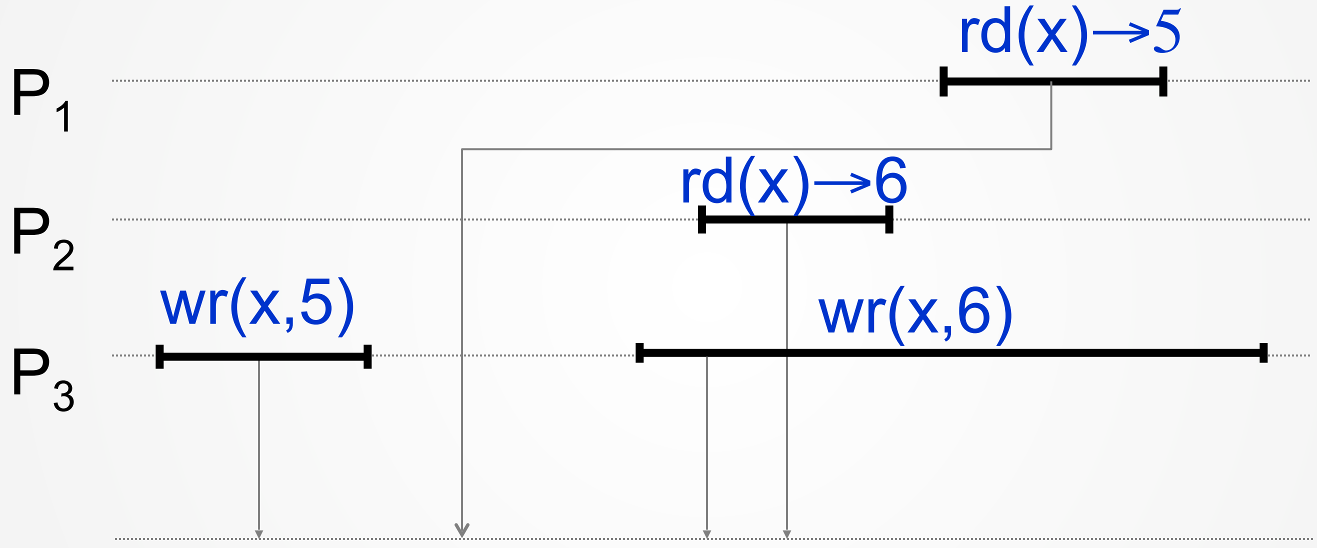

Linearization points
Single System Image

# Example 2




Linearization points
Single System Image

# EXAMPLE 3 SEQUENTIAL CONSISTENCY



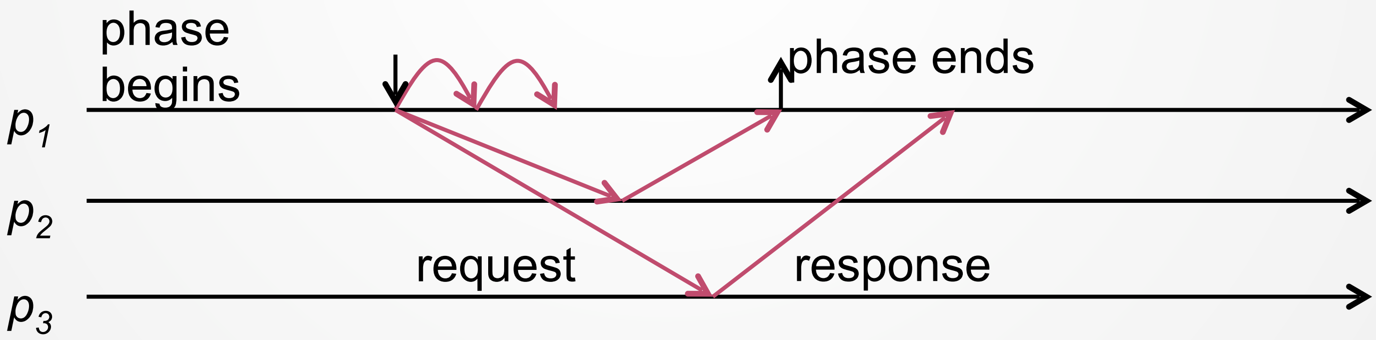

Sequential Execution

# (1,N) Algorithm [Fail-Silent]

# PHASES

A phase run by $p_i$ consists of:

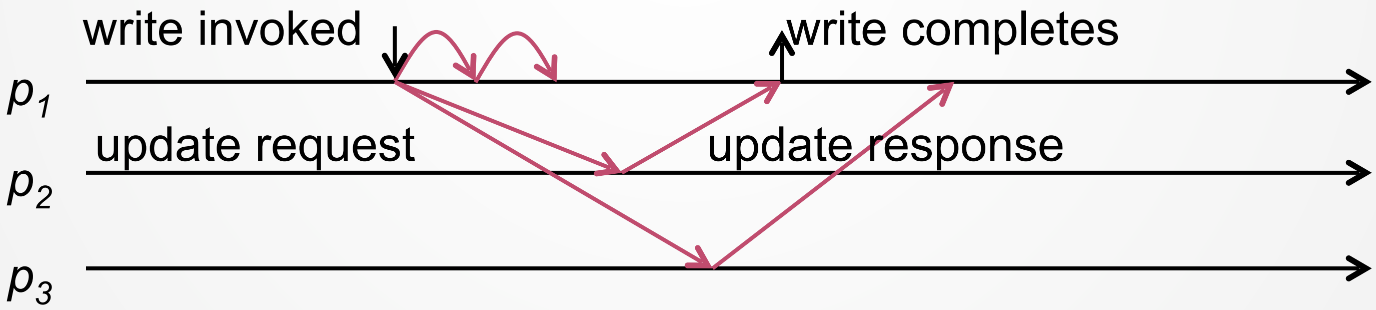
- $p_i$ beb-broadcasts a request
- $p_j$ receives request, processes it, and sends response
- $p_i$ waits for responses from a majority before the phase ends



# WRITE OPERATION MAJORITY VOTING

Writer executing write($r$, $v$) operation

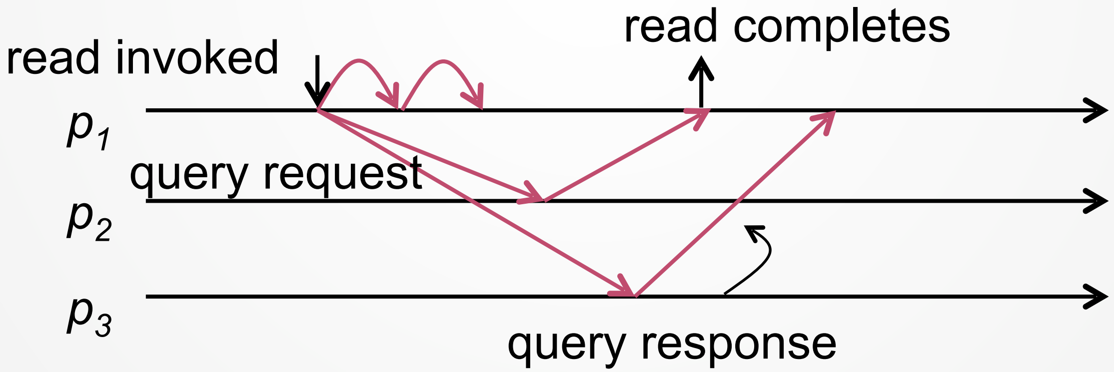
- $ts$++ (increment current sequence number)
- $p_i$ forms $tvp$=(ts, $v$), where $ts$ is current sequence number
- $p_i$ starts an ***update phase*** by sending **update request** with register id $r$ and ts pair (ts, v)
- $p_j$ updates r = **max(r, (ts, v))** and responds with ACK
- $p_i$ completes write when update phase ends



# READ OPERATION MAJORITY VOTING

Process $p_i$ executing read($r$) operation

- $p_i$ starts **query phase**, sends query request with id $r$
- $p_j$ responds to the query with $(\text{ts}, \text{v})_j$
- When query phase ends, $p_i$ picks **max $(\text{ts}, \text{v})_j$ received**



# MAJORITY VOTING ALGORITHM (1,N)

Assume majority of ***correct processes***

Register values have a sequence number (seq#)

No FD

to **write(v)**

**ts**++

Broadcast **v** and **ts** to all

if newer **ts**:

Receiver update to (**ts, v**)

Receiver sends ACK

Wait for ACK from ***majority of nodes***

Return

The **update phase** with (v,ts)

to **read**

Broadcast read request to all

Receiver respond with local value **v** and **ts**

Wait and save values from ***majority of nodes***

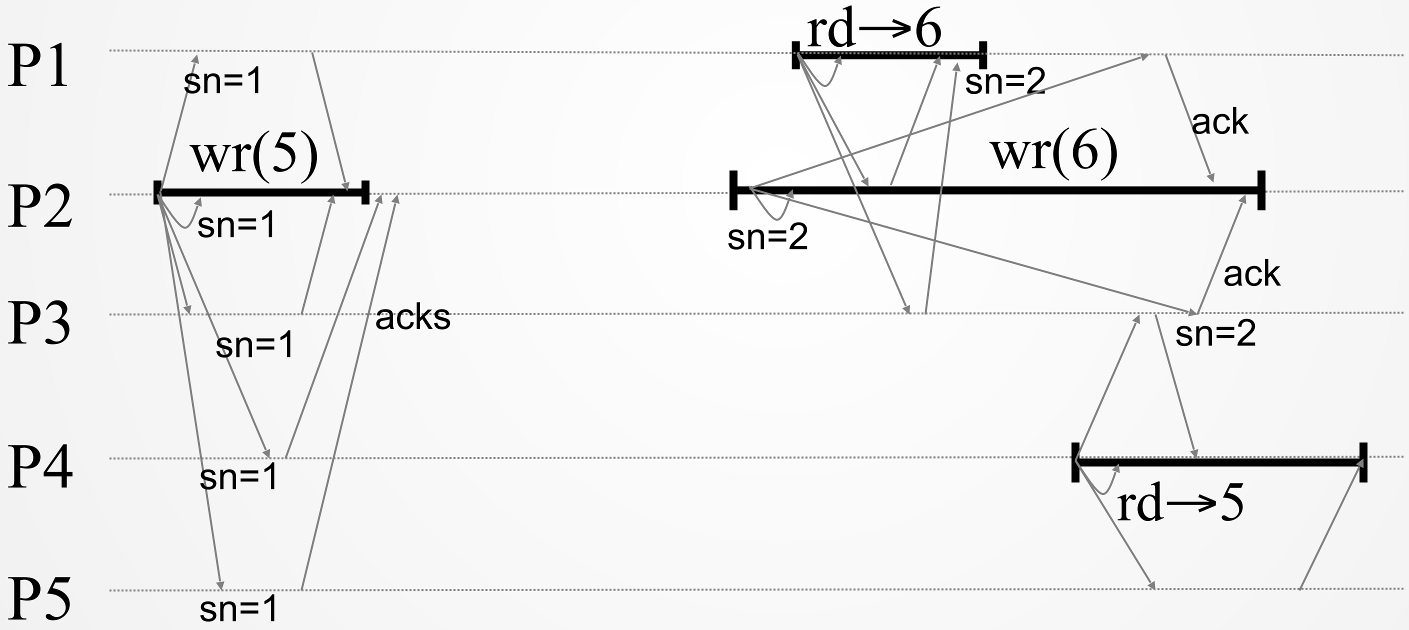
Return value with ***highest*** **ts**

The read **query phase**

# Regular but not Atomic

Problem with majority voting

Ex: majority(5)=3

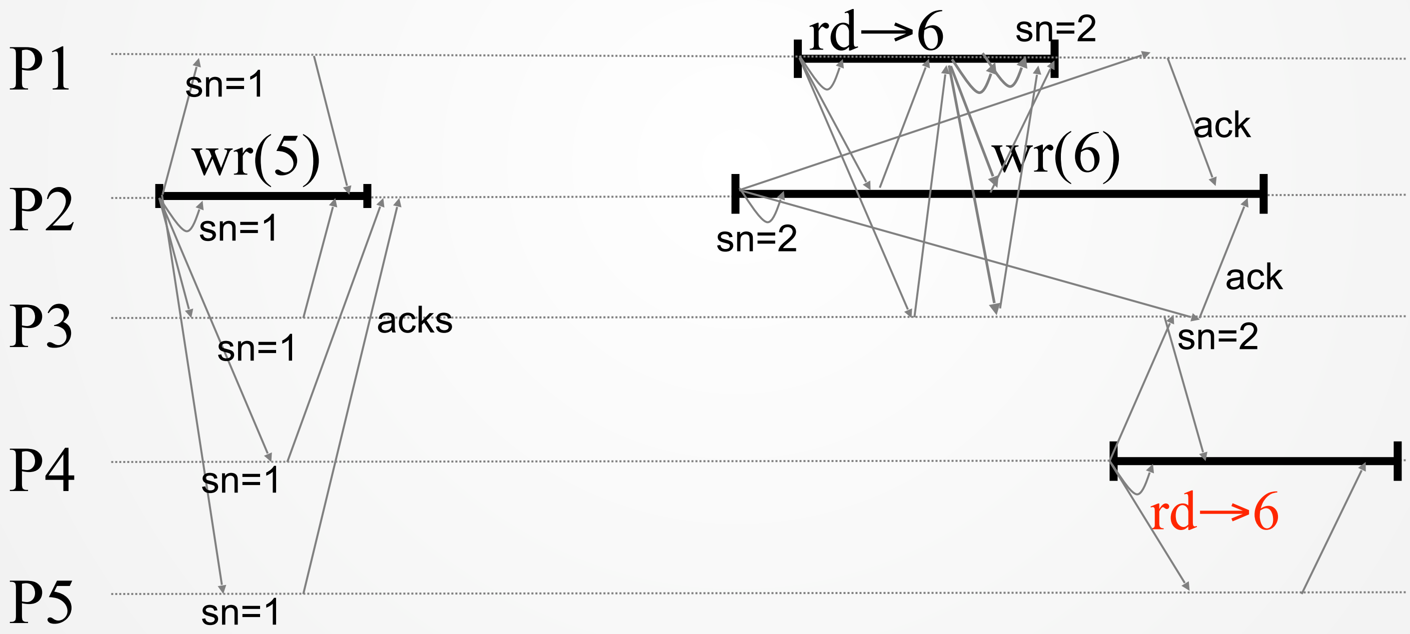


# READ IMPOSE

***Main idea***

***Read-impose (update)***

***When reading, also do an update before responding***



# READ-IMPOSE WRITE MAJORITY (1,N)

to **read**

Broadcast read request to all

Receiver respond with local value **v** and **ts** ← query phase

Wait and save values from ***majority of nodes***

Perform an **update phase** with ***highest*** (**ts, v**)

Return value **v**

- **Optimization**
  - if all responses in the query phase have the same **ts** do not perform the update phase, just return
- A majority has the latest value written

# WHY DOES IT WORK? WHY READ-IMPOSE

Validity

- Read returns ***last value written*** if
  Not ***concurrent*** with another write
  Not concurrent with a ***failed operation***
- Otherwise may return last or concurrent "value"

Ordering

- If a **read→r1** precedes **read→r2**
- Then **write(r1)** precedes **write(r2)**

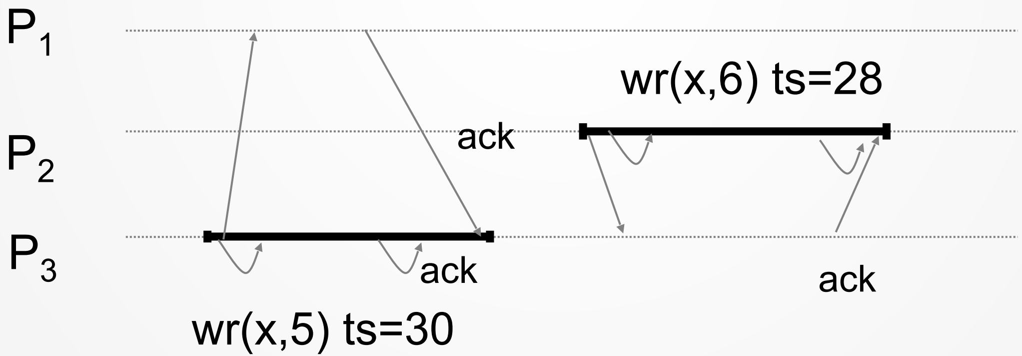
- A read rd(x)⇒r1 makes an update with r1
- Any succeeding read must at least see r1

- ***Causality used to enforce atomicity***

# (N,N) Algorithm [Fail-Silent]

# ATOMIC REGISTER (MULTIPLE WRITERS)

- Read-Impose Majority Voting
  - Multiple writers might have non-synchronized time stamp **ts**
- Example:
  - The latter wr(x, 6) is ignored because old timestamp



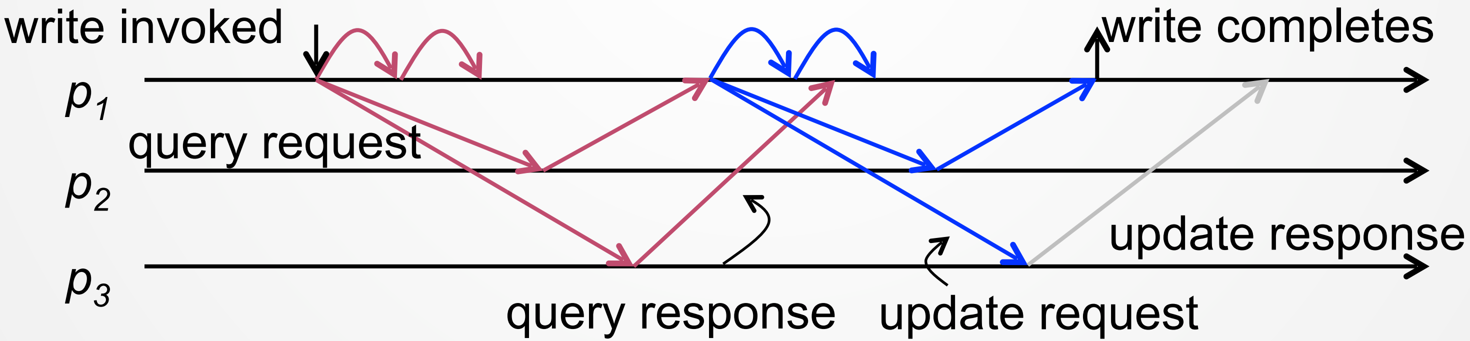
# ATOMIC REGISTERS (N,N) 1/2

- ***Read-impose write-consult-majority (N,N)***
  - Before writing, read from majority to get last ts
  - Do a query phase to get the latest timestamp before the update phase

- Problem
  - Two concurrent writes with same timestamp?
  - Just compare process identifier, break ties!
  - Initially the value of register $X_r$ of $p_i$ is $((0,i),\perp)$

# WRITE OPERATION — QUERY PHASE

- Process $p_i$ executing operation wr($X_r$, v)
  - $p_i$ starts **query phase**, sends query request with id *r*
  - $p_j$ responds to the query with current timestamp $(ts, pid)_r$
- When query phase ends, $p_i$ picks **max $(ts, pid')_r$** received
  - $p_i$ starts an ***update phase*** by sending update request with register id *r* and timestamp-value pair **((ts+1, i), v)**
  - $p_j$ updates r = **max(r, ((ts, pid), v))** and responds with ACK
  - $p_i$ completes write when update phase ends



# Atomic Registers (N,N) 2/2

- ***Read-impose write-consult-majority (N,N)***
  - **update phase**
    - Before writing, read from majority to get last timestamp

Wait-free: Every correct process should "make progress"
(no deadlocks, no live-locks, no starvation)

- Observe in all phases, any process $p_i$ sends ACK message even if $p_i$ receives update request with old timestamp
  - Because of multiple writers
  - Example:
    - Slow P1 does update(x, (5), waits for acks
    - Fast P2 writes(6), receives acks from majority
    - P1 does not get enough acks, as nodes ignore its write(5)
    - P1 stalls

# ATOMIC REGISTER (N,N) SUMMARY

- For atomic register
  - A write to complete requires 2 round-trips of messages
    - One for the timestamp (query phase)
    - One for broadcast-ACK (update phase)
  - A read to complete  requires 2 round-trips of messages is
    - One for read (query phase)
    - One for impose if necessary (update phase)

# (N,N) algorithm

# Proof of linearizability

# LINEARIZABILITY (LIN)

- LIN($T$) requires that there exists legal history $S$:
  - $S$ is equivalent to T,
  - **If $o_1 <_T o_2$ then it must also be that $o_1 <_S o_2$**

- LIN is compositional: $(\forall x_r\text{: LIN}(T|x_r)) \Leftrightarrow \text{LIN}(T)$

- We focus on arbitrary register $X_r$ and proof LIN(T|$x_r$)

ID2203

KTH-2023

# LEGAL SEQUENTIAL ORDER

- Timestamp of operation $o$, $ts(o)$, is timestamp used in $o$'s update phase of the write and read operations
- Construct $S$ from $T|x_r$ in timestamp order:
  1. Order writes $o_w$ according to their (unique) timestamp (ts,i)
  2. Order each read $o_r$ immediately after write with same time stamp (ts, i)
     - For reads with same ts, order them by increasing invocation order in the (real time) trace
- $S$ is legal by construction
  - S is sequential and read returns last value written

# COMPLETING THE PROOF

We must show that, for each execution, and for each register $x_r$, LIN($T \mid x_r$) holds

- Requires that there exists legal history $S$ s.t.
  - $S$ is equivalent to $T|x_r$,
  - $S$ preserves order of non-overlapping ops in $T|x_r$

# EQUIVALENCE

➡ S preserves non-overlapping order as $T|x_r$

- $S$ and $T|x_r$ are equivalent
    - They contain same events
    - $(T|x_r)|p_i$ contains non-overlapping operations
    - $(T|x_r)|p_i = S|p_i$

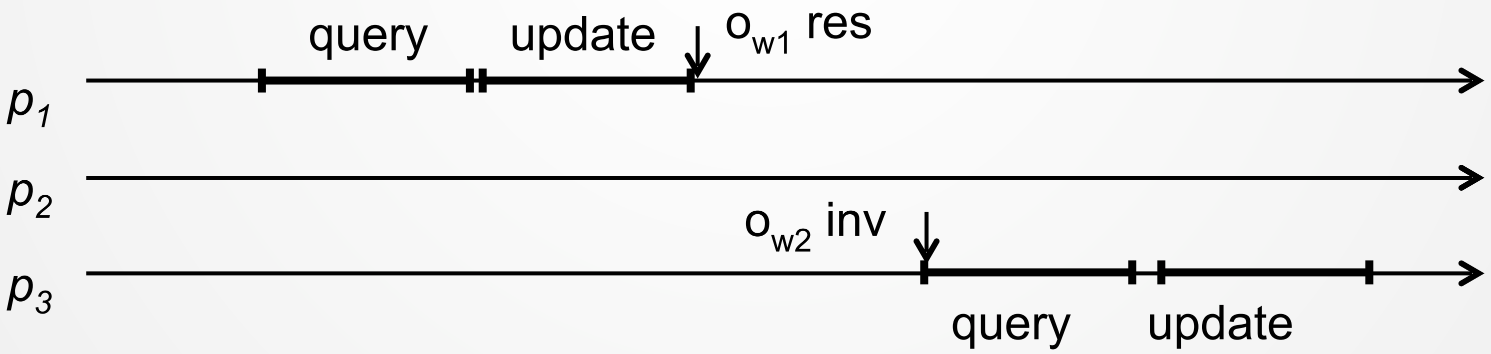
- Hence, LIN($T|x_r$) for any register $x_r$, which implies LIN($T$)

# PRESERVING NON-OVERLAPPING ORDER

- Must show that $S$ preserves the order of non-overlapping ops in $T|x_r = T'$
  - If $o_1 <_T o_2$ then it must also be that $o_1 <_S o_2$
  - $res(o_1) <_{T'} inv(o_2) \Rightarrow res(o_1) <_s inv(o_2)$

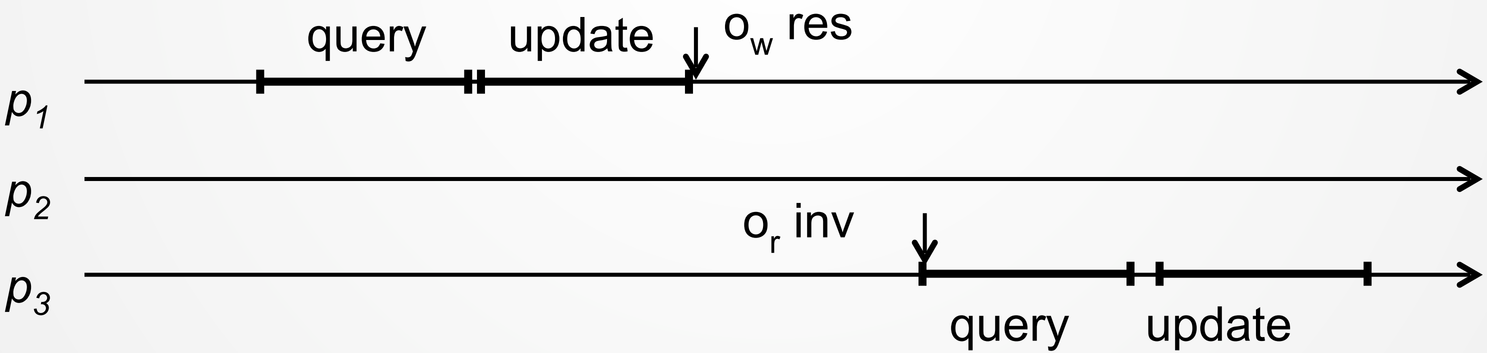
# O1 AND O2 ARE WRITE OPERATIONS

- $o_{w1} <_{H'} o_{w2} \Rightarrow o_{w1} <_s o_{w2}$
- $res(o_{w1}) <_{H'} inv(o_{w2}) \Rightarrow \mathrm{ts}(o_{w1}) < \mathrm{ts}(o_{w2})$
- $o_{w1}$ update phase is before $o_{w2}$ *query phase*
- $o_{w2}$ query returns a timestamp $\geq \mathrm{ts}(o_{w1})$
- $o_{w2}$ increments the timestamp
- Hence $\mathrm{ts}(o_{w1}) < \mathrm{ts}(o_{w2}) \Rightarrow o_{w1} <_s o_{w2}$



# O1 (OW ) WRITE AND O2 (OR) IS READ

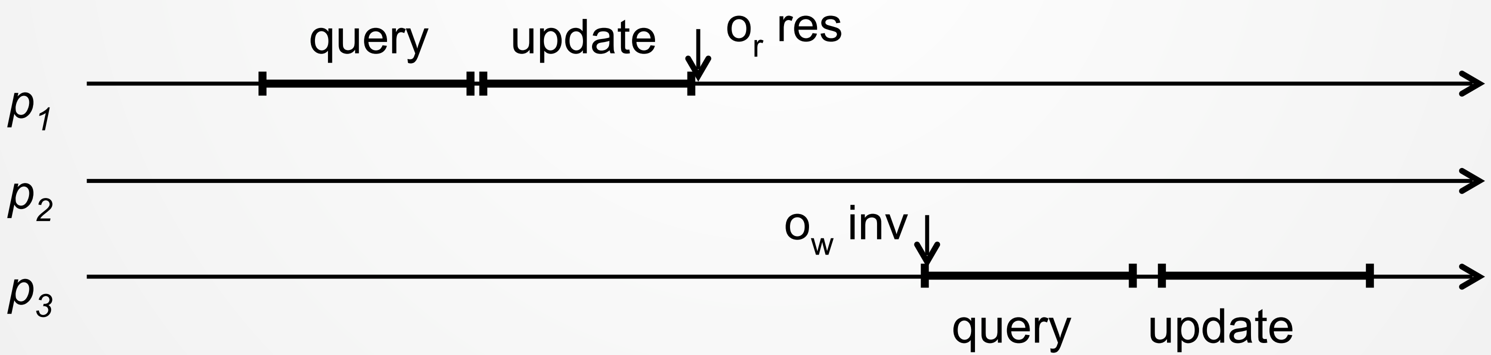
- $o_w <_{H'} o_r \Rightarrow o_w <_s o_r$
- $res(o_w) <_{H'} inv(o_r) \Rightarrow \mathrm{ts}(o_w) \leq \mathrm{ts}(o_r)$
- $o_w$ update phase is before $o_r$ *query phase*
- $o_r$ returns a timestamp $\geq$ $\mathrm{ts}(o_w)$
- Hence $o_w <_s o_r$




ID2203
KTH-2023

# O1 (OR) IS READ AND O2 (OW) IS WRITE

- $o_r <_{H'} o_w \Rightarrow o_r <_s o_w$
- $res(o_r) <_{H'} inv(o_w) \Rightarrow \mathrm{ts}(o_r) < \mathrm{ts}(o_w)$
- $o_r$ update phase is before $o_w$ *query phase*
- $o_w$ *query phase* returns a timestamp $\geq \mathrm{ts}(o_r)$
- $o_w$ increments the timestamp
- Hence $\mathrm{ts}(o_r) < \mathrm{ts}(o_w) \Rightarrow \mathrm{ts}(o_r) < \mathrm{ts}(o_w)$



# O1 (OR1) IS READ AND O2 (OR2) IS READ

- $o_{r1} <_{H'} o_{r2} \Rightarrow o_{r1} <_s o_{r2}$
- $res(o_{r1}) <_{H'} inv(o_{r2}) \Rightarrow$
  $\mathrm{ts}(o_{r1}) < \mathrm{ts}(o_{r2})$ or $(\mathrm{ts}(o_{r1}) = \mathrm{ts}(o_{r2})$ and $\mathrm{inv}(o_{r1}) <_{H'} \mathrm{inv}(o_{r2}))$
- $o_{r1}$ update phase is before $o_{r2}$ *query phase*
- $o_{r2}$ *query* returns a timestamp $\mathrm{ts}(o_{r2}) \geq \mathrm{ts}(o_{r1})$
- if $\mathrm{ts}(o_{r1}) < \mathrm{ts}(o_{r2})$ then $o_{r1} <_s o_{r2}$ (at least one $o_w$ in between)
- if $\mathrm{ts}(o_{r1}) = \mathrm{ts}(o_{r2})$ then $\mathrm{inv}(o_{r1}) <_{H'} \mathrm{res}(o_{r1}) <_{H'} \mathrm{inv}(o_{r2})$
  - Hence $o_{r1} <_s o_{r2}$