Shared Memory

Seif Haridi

haridi@kth.se

Real Shared Memory

- Formal model of shared memory
  - No message passing (No channels, no sends, no delivers of messages)
  - Instead processes access a shared memory
  - Models multiprocessors, multicores…
- We are interested in distributed systems
  - Implement (simulate) a distributed shared memory using message passing

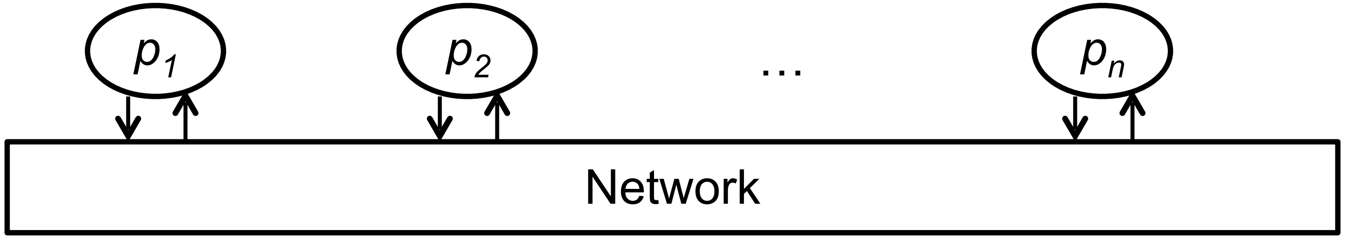
Distributed Shared Memory

- “Simulate” that the DS has a ***shared memory***
  - A ***register*** represents each memory location
    - Registers aka objects
  - processes can ***read***/***write*** to a set of registers
    - Not only RW-registers… 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

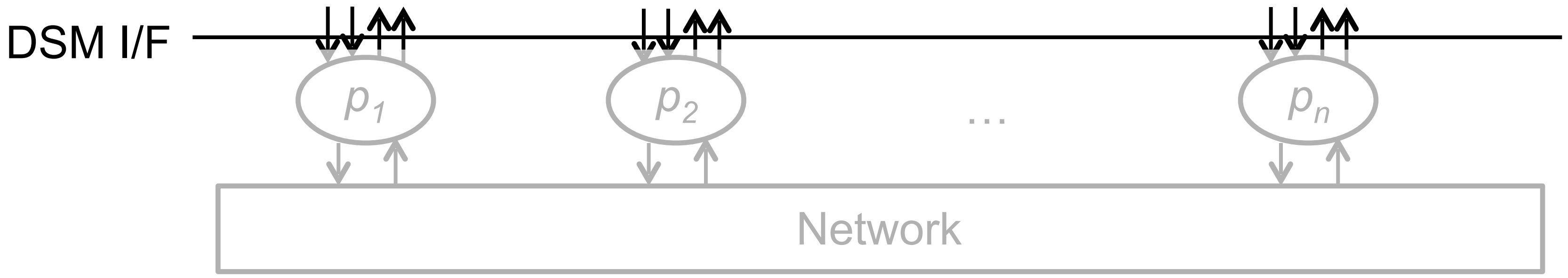


Read/Write Register

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

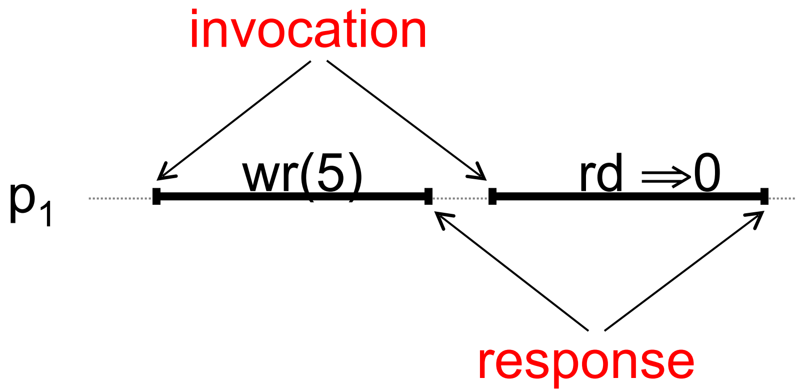
Distributed Shared Memory

- 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



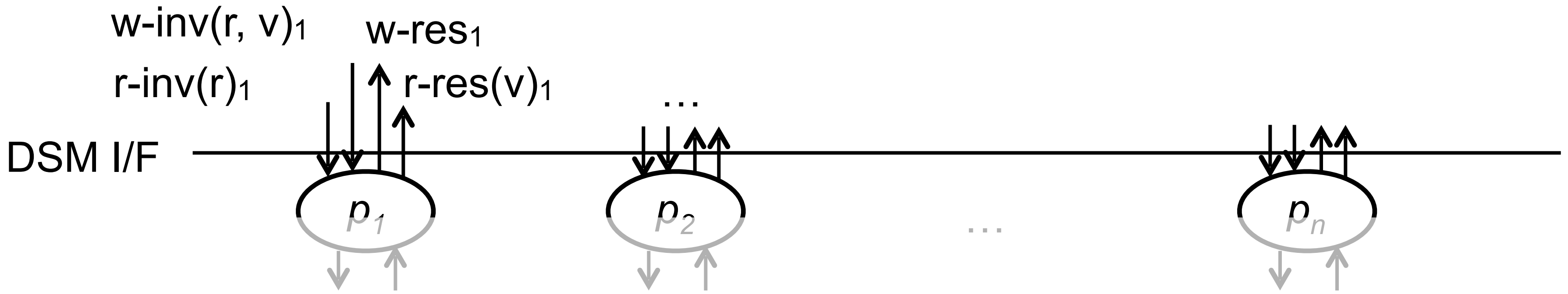
Read/Write Register

- 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



DSM interface

- Interface events:
  - r-inv$_i$(r) – invoke read of x_r by p_i
  - r-res$_i$(v) – response to read with value v by p_i
  - w-inv$_i$(r, v) – invoke write with value v to x_r by p_i
  - w-res$_i$ – response to write by p_i



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
  - $r\text{-}inv_i(r)$
    - Read invocation by process pi on register X_r
  - $r\text{-}res_i(v)$
    - Response with value v to read by process pi
  - $w\text{-}inv_i(r,v)$
    - Write invocation by process pi on register X_r with value v
  - $w\text{-}res_i$
    - Response (confirmation) to write by process pi

Trace properties

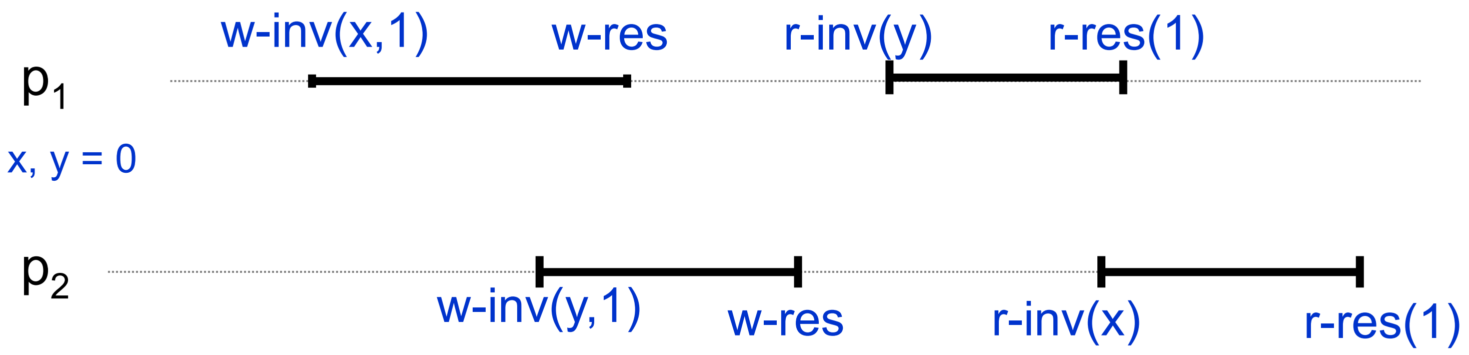
- Trace is **well-formed**
  - First event of every process is an invocation
  - Each process alternates between invocations and responses
- Trace is **sequential** if
  - x-inv by i immediately followed by a *corresponding* x-res at i
  - x-res by i immediately follows a *corresponding* x-inv by i
  - 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

Definitions

- In a trace *T*, an operation O 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

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

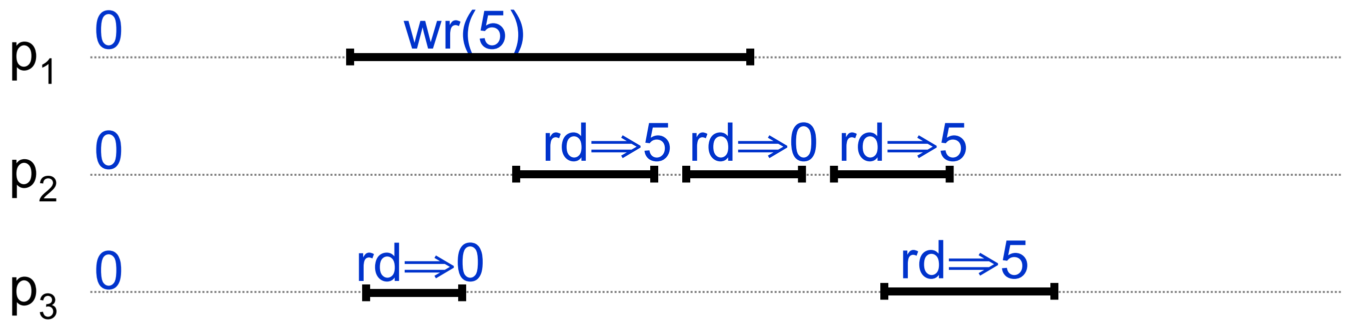
- (1,N)-algorithm
  - 1 designated writer, multiple readers
- (M,N)-algorithm
  - Multiple writers, multiple readers

Regular Register (1, N)

- Termination
  - Each read and write operation of a correct node 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?*** [d]
  - Does not work if leader crashes

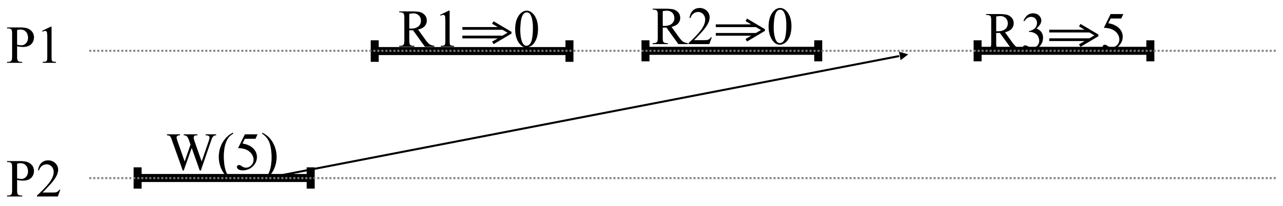
Bogus Algorithm (regular)

- 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?*** [d]



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

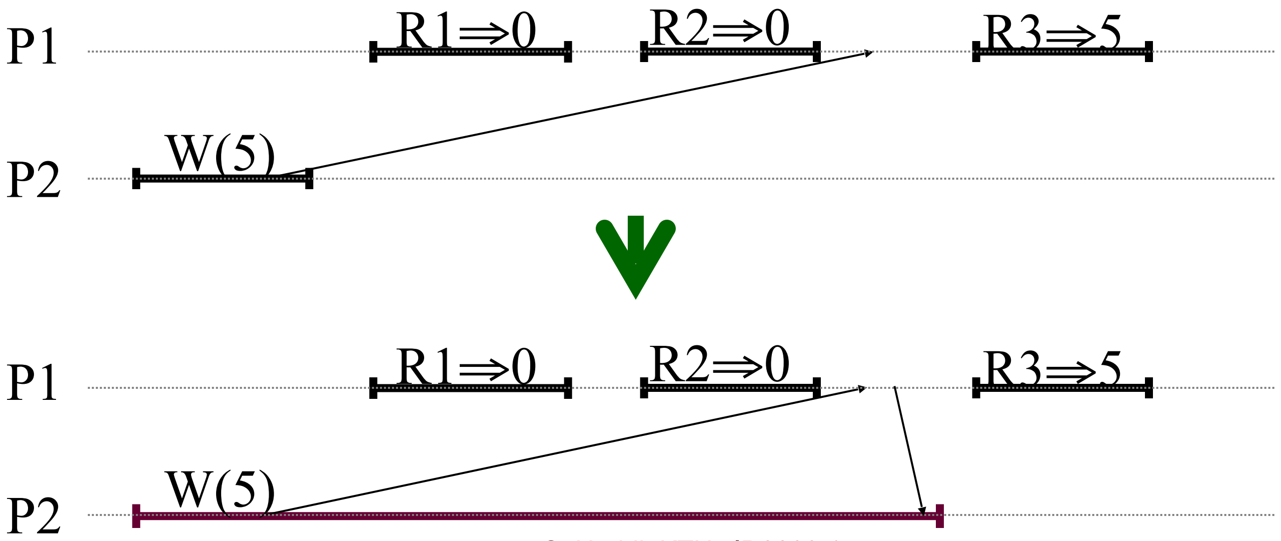
Correctness

- 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
    - At a read is invoked each process either stores

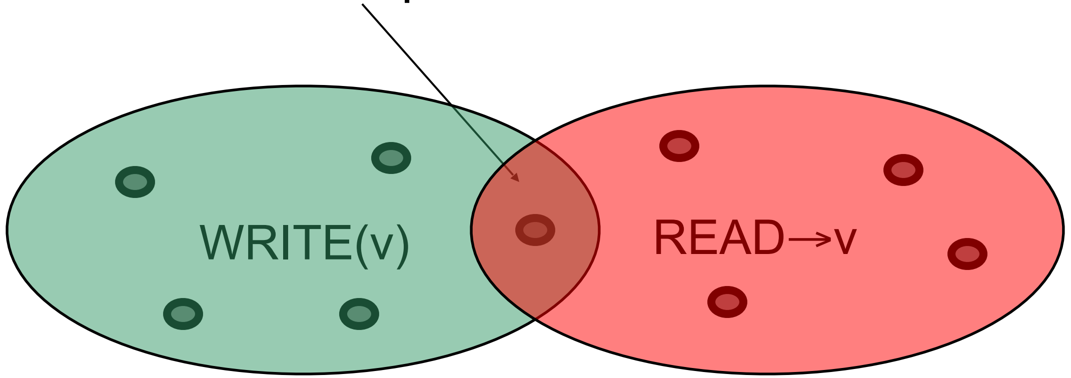
Read-one Write-All (1,N) #2

- ***Main idea***
  - Postpone write responses



Majority Voting Algorithm Fail-Silent model

- ***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

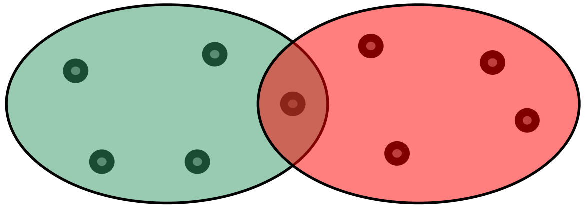


- Ex: majority(9)=5

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

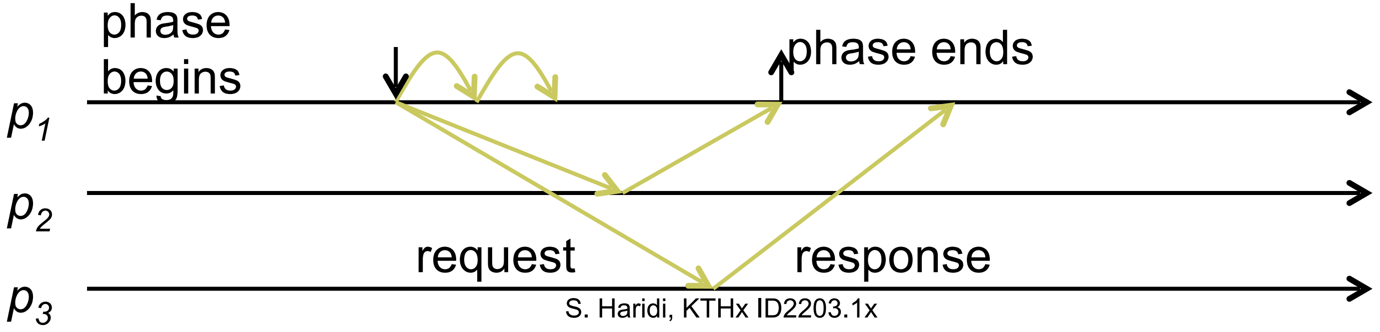


Timestamp-value pairs

- 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, ⊥) at all processes
- Each process
  - Stores the value of register r with max timestamp for each register r

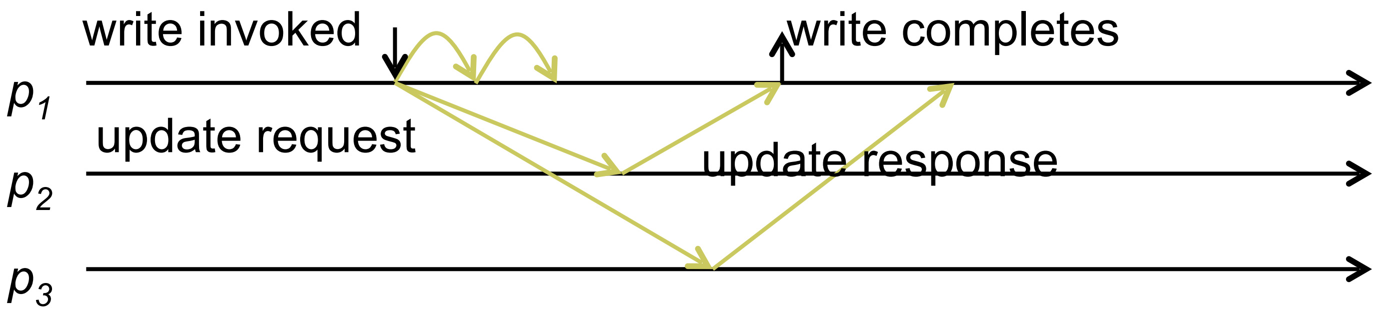
Phases

- The communication involved in operations are structured 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



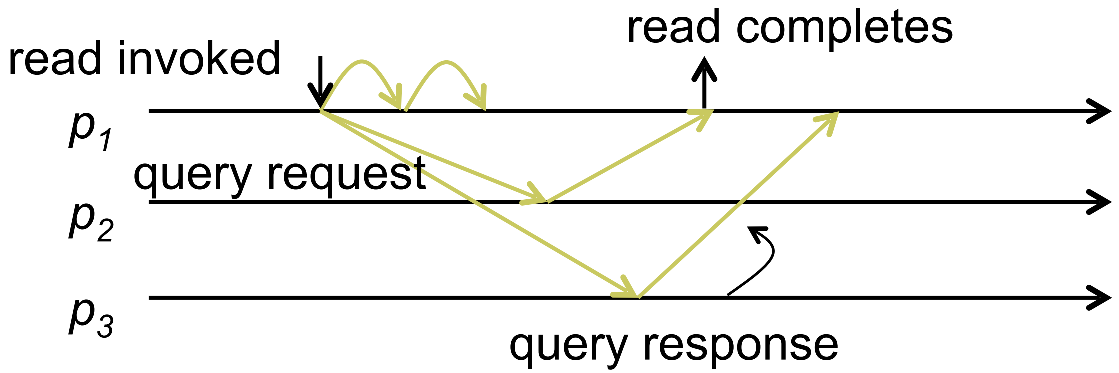
Write operation Majority Voting

- 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 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



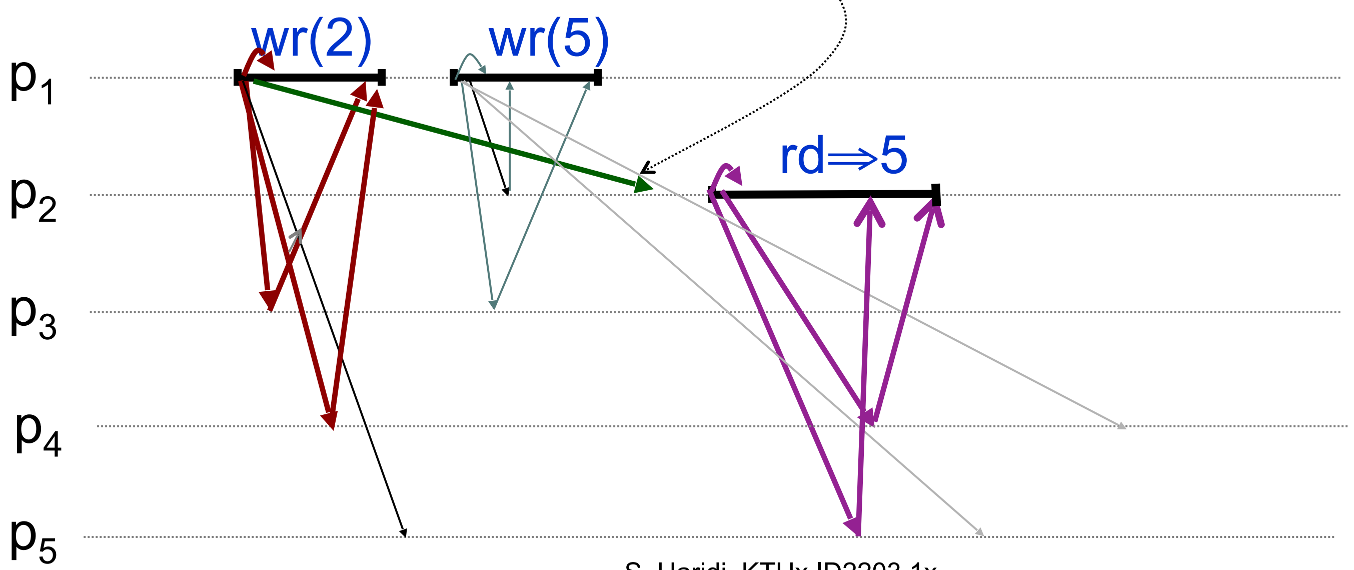
Read operation Majority Voting

- 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 (ts, v)$_j$
  - When query phase ends, p_i picks **max (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

1. **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
2. **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

- Safety requirements
  - Sequential Consistency
    - Informally:

      only allow executions whose results appear as if there is a single system image and "local time" is obeyed

  - Linearizability/Atomicity
    - Informally:

      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 X_r returns last value written to register X_r
- Given a trace T, **T|*p*$_i$** (view of process p_i)
  - Subsequence of T with only x-inv$_i$ and x-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

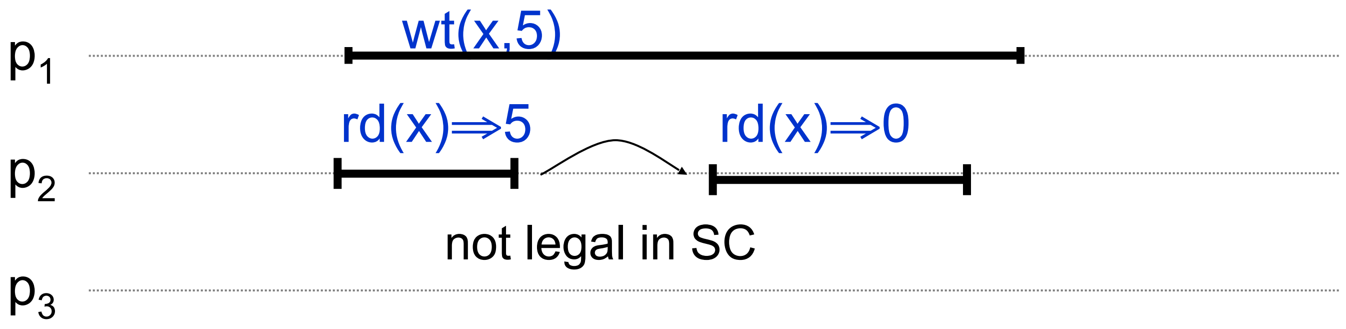
- LIN is a consistency condition similar to SC
  - LIN(*T*) requires that there exists legal Trace *S*:
    - *S* is equivalent to *T*,
    - **If $o_1 <_T o_2$ then it must also be that $o_1 <_S o_2$**
- LIN is stronger than SC: LIN(*T*) $\Rightarrow$ SC(*T*)

Failure for Linearizability and Sequential Consistency

- 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)

Motivating Example 1

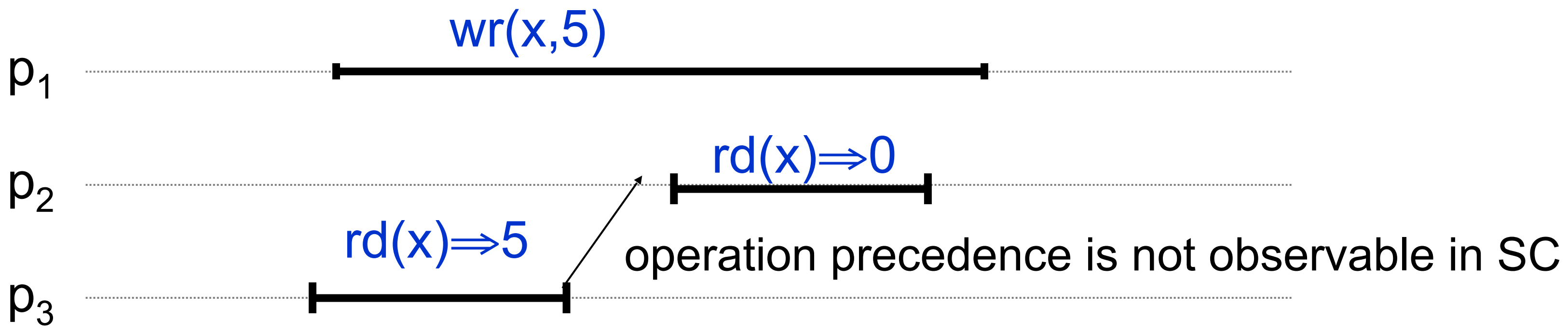
- Regular execution



- Sequential consistency disallows such E's

Motivating Example 2

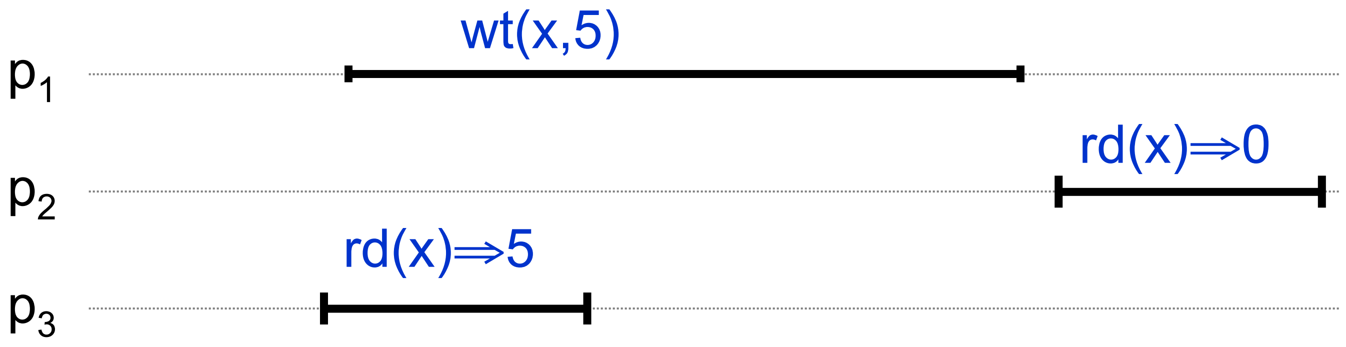
- Regular execution



- Sequential consistency allows such T's

Motivating Example 2

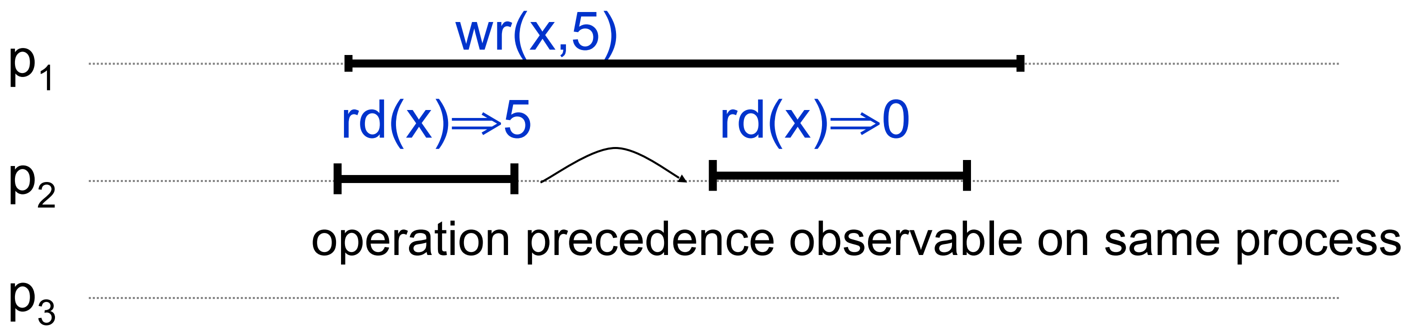
- Sequentially consistent execution



- Regular consistency disallows such trace

Motivating Example 1

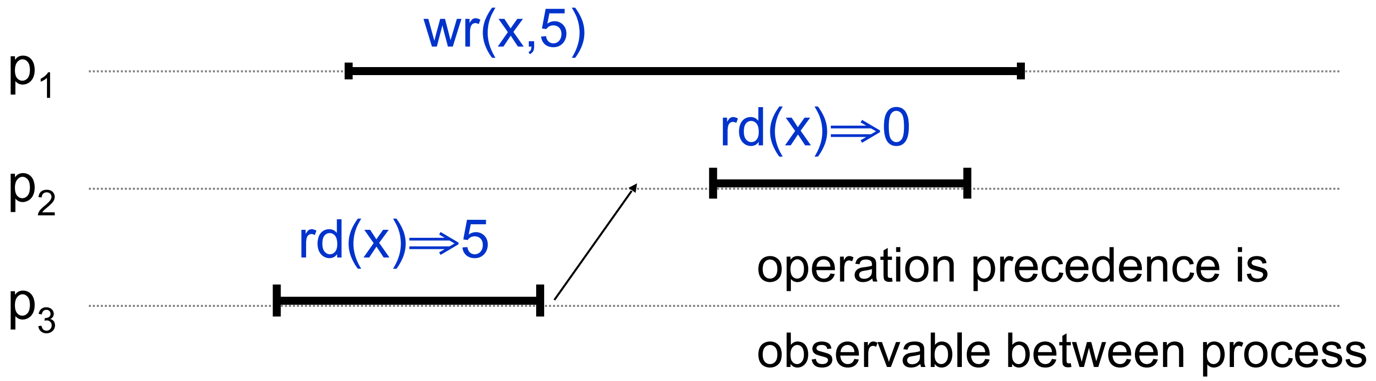
- Regular execution



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

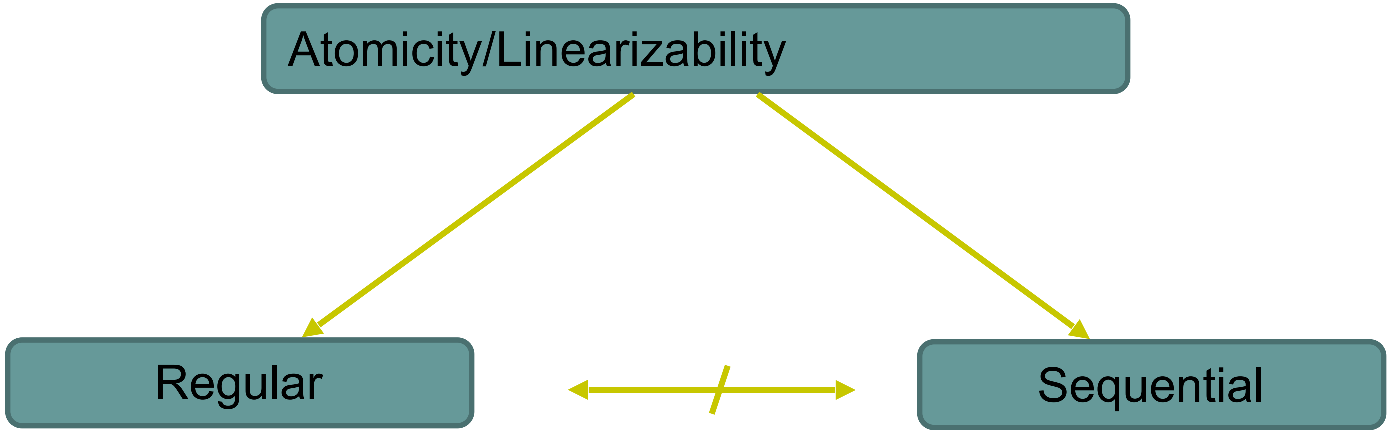
Motivating Example 2

- Regular execution



- Atomicity/Linearizability disallows such E's

Consistency hierarchy



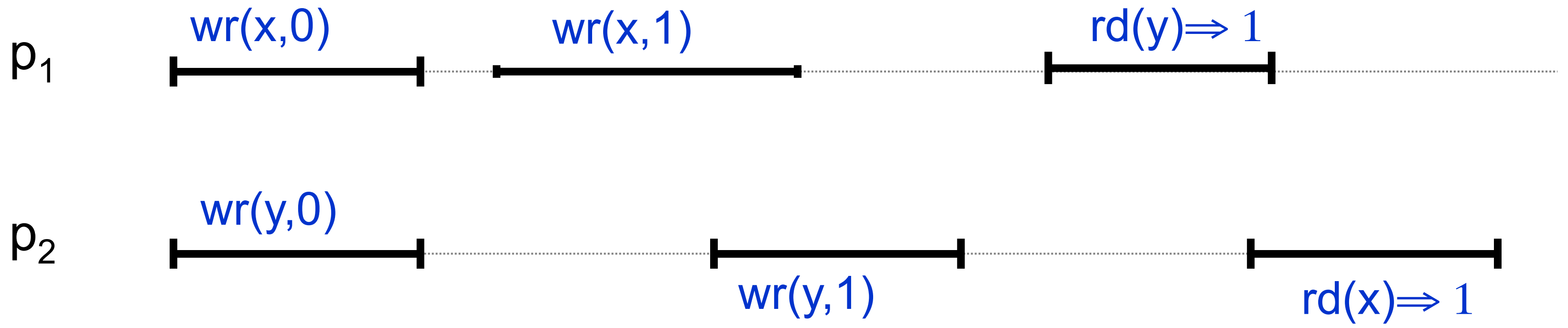
Compositionality

- For a trace T
  - T | x_r Subsequence of T with only x-inv and x-res of register 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 | x_r)) $\Leftrightarrow$ CC(T)

Compositionality

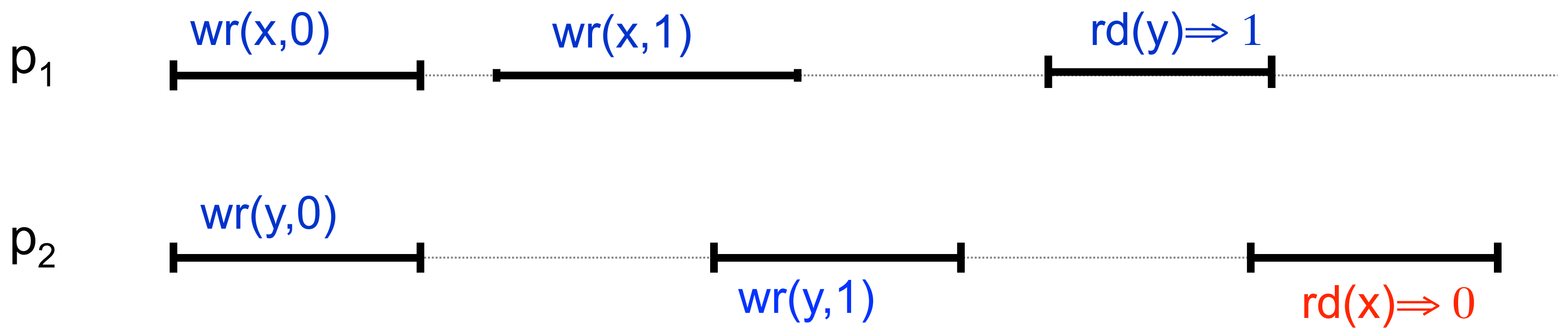
- Possible with compositional consistency condition
  - Consistency condition CC(H) is compositional iff
    - $(\forall x_r: CC(H|x_r)) \Leftrightarrow CC(H)$
- Linearizability is compositional
  - for all registers x_r: $LIN(T|x_r) \Leftrightarrow LIN(T)$
- Unfortunately, SC is not compositional
 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

Example Linearizable Trace



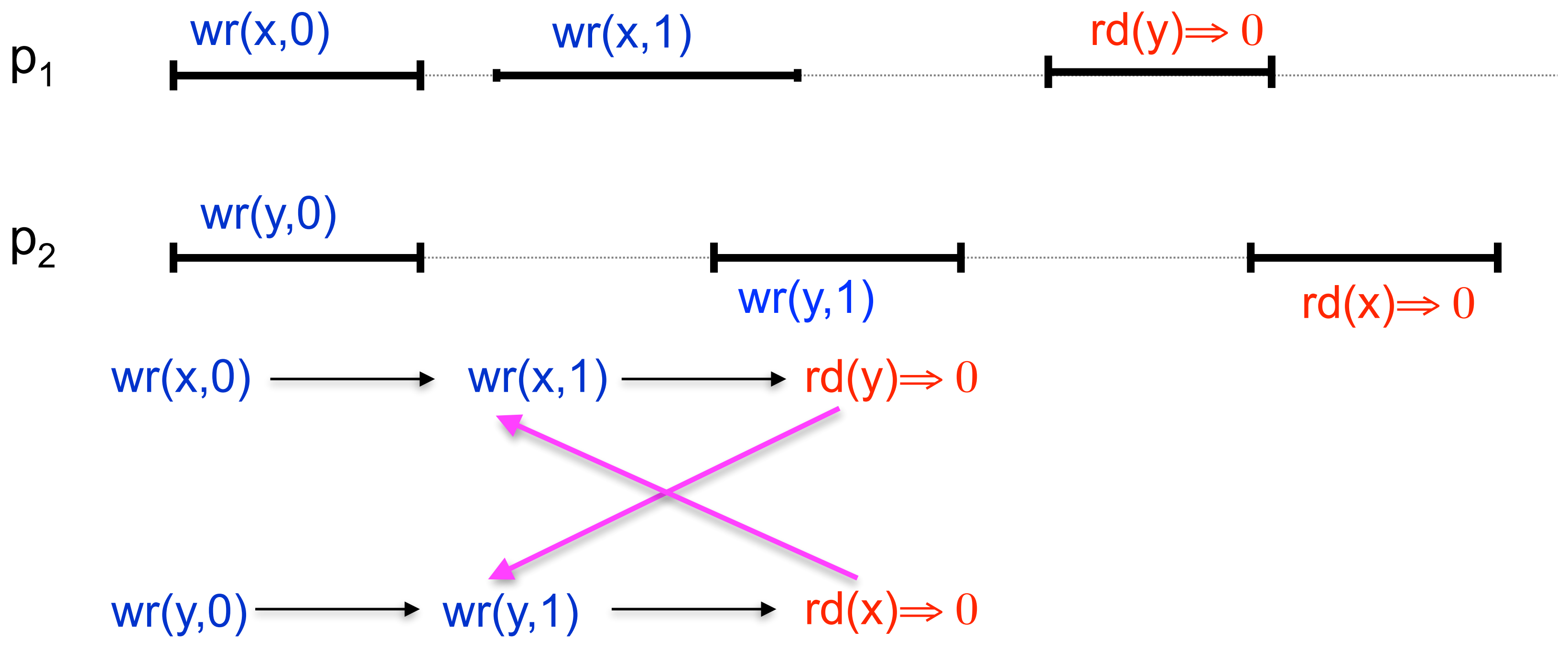
wr(x,0) wr(y,0) wr(y,1) wr(x,1) rd(y)⇒ 1 rd(x)⇒ 1

Example Sequentially Consistent Trace

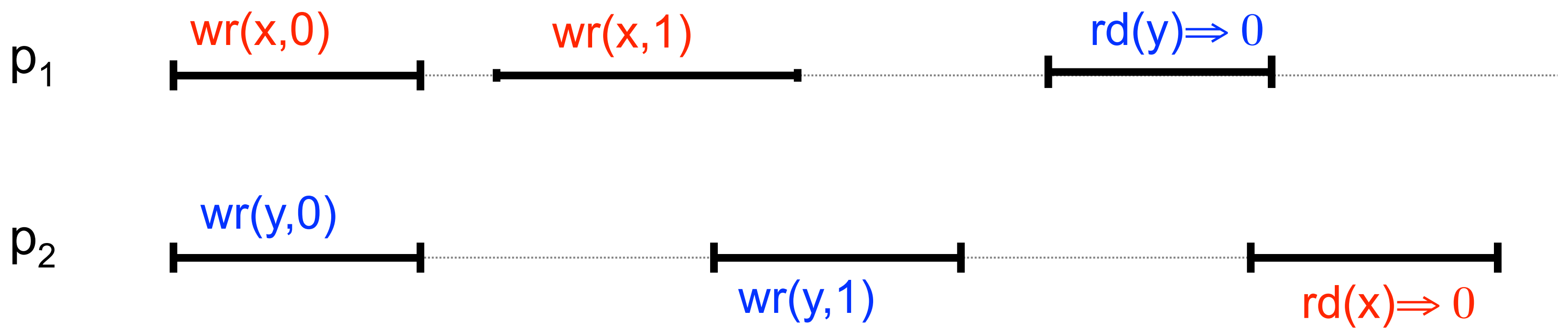


wr(x,0) wr(y,0) wr(y,1) rd(x)⇒ 0 wr(x,1) rd(y)⇒ 1

Not Sequentially Consistent Trace



Sequential Consistent is not Compositional



$$\text{wr}(x,0) \longrightarrow \text{rd}(x) \Rightarrow 0 \longrightarrow \text{wr}(x,1)$$

$$\text{wr}(y,0) \longrightarrow \text{rd}(y) \Rightarrow 0 \longrightarrow \text{wr}(y,1)$$

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

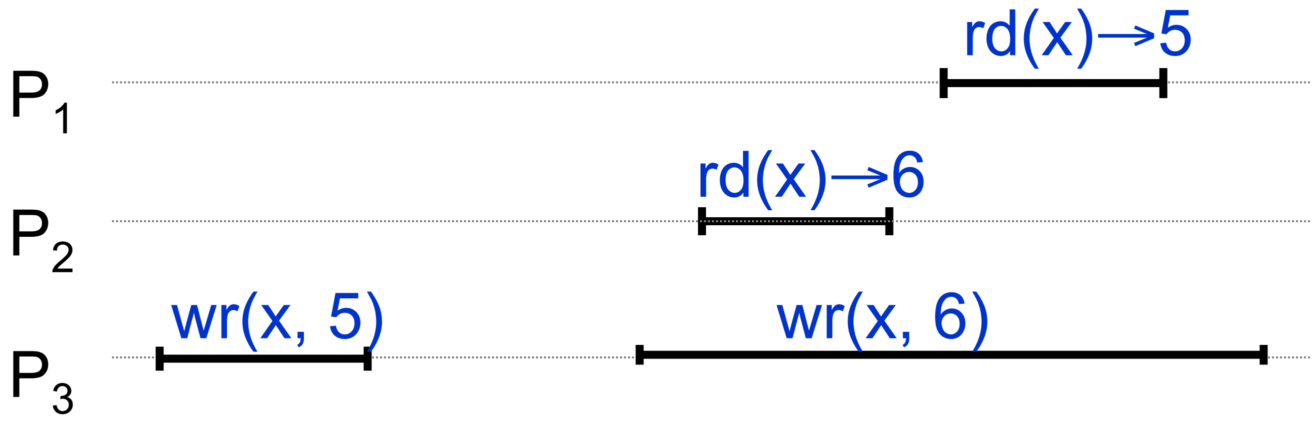
- **Read ops** appear as **immediately** happened at all nodes at
  - time between invocation and response
- **Write ops** appear as **immediately** happened at all nodes at
  - time between invocation and response
- **Failed ops** appear as
  - completed at every node, XOR
  - never happened at any node

same ⟷

Ordering (only (1,N))

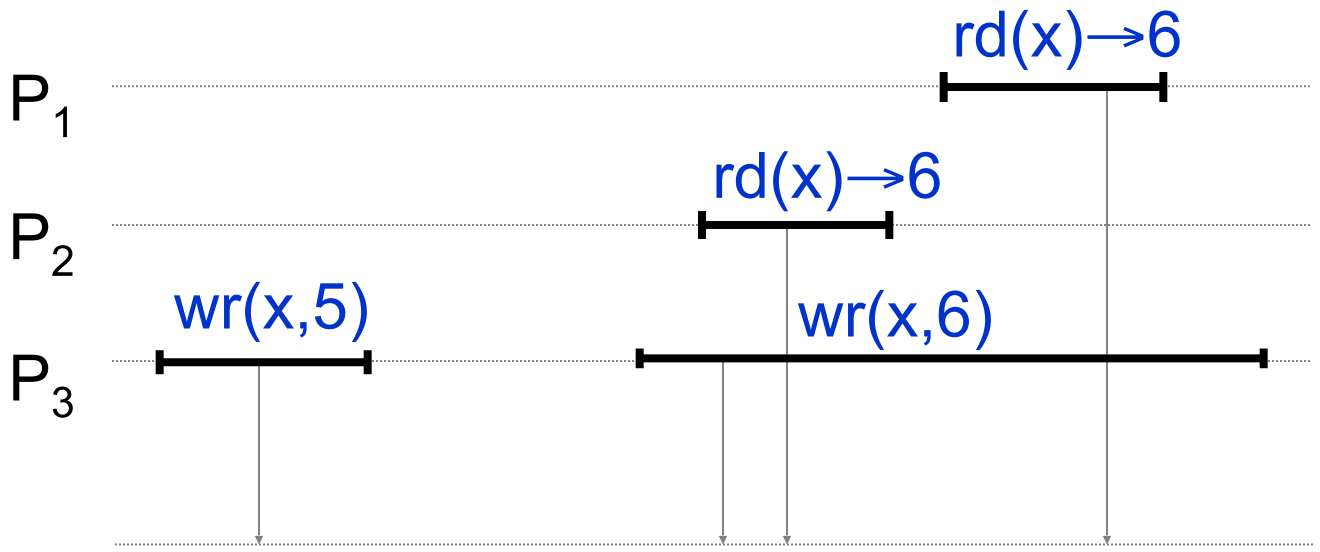
- Validity
  - Read returns last value written if
    - Not concurrent with another write
    - Not concurrent with a failed operation
  - Otherwise may return last or concurrent "value"
- Ordering
  - If read→r1 precedes read→r2 then write(r1) precedes write(r2)

Example



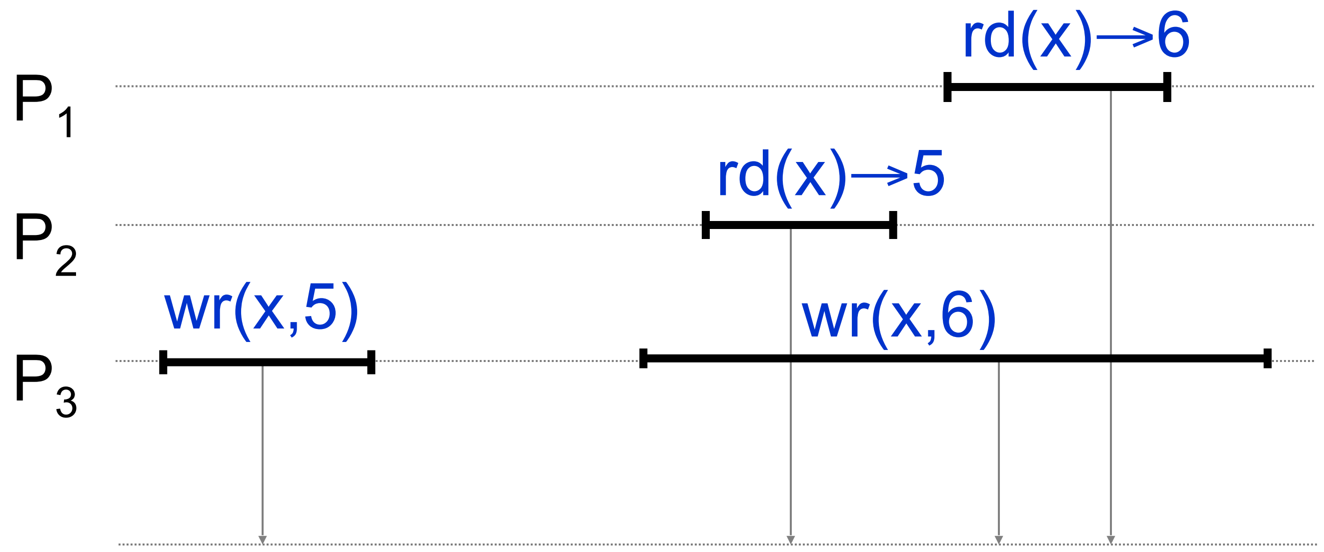
- Atomic? [d]
  - No, not possible to find linearization points

Example 2



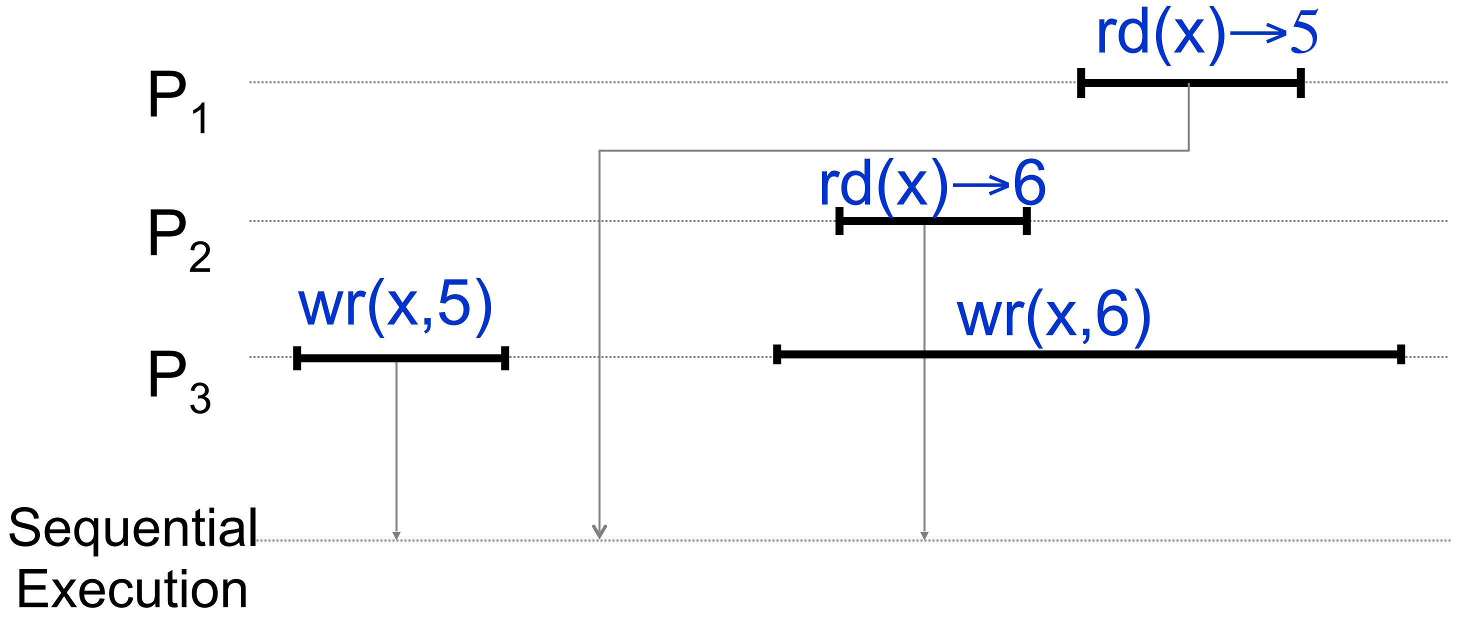
Linearization points
Single System Image

Example 2



Linearization points
Single System Image

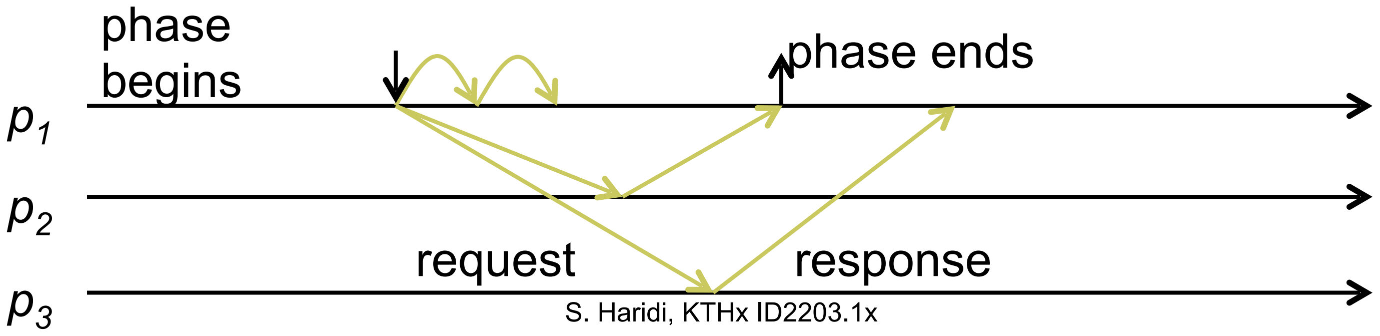
Example 3 Sequential Consistency



(1,N) Algorithm

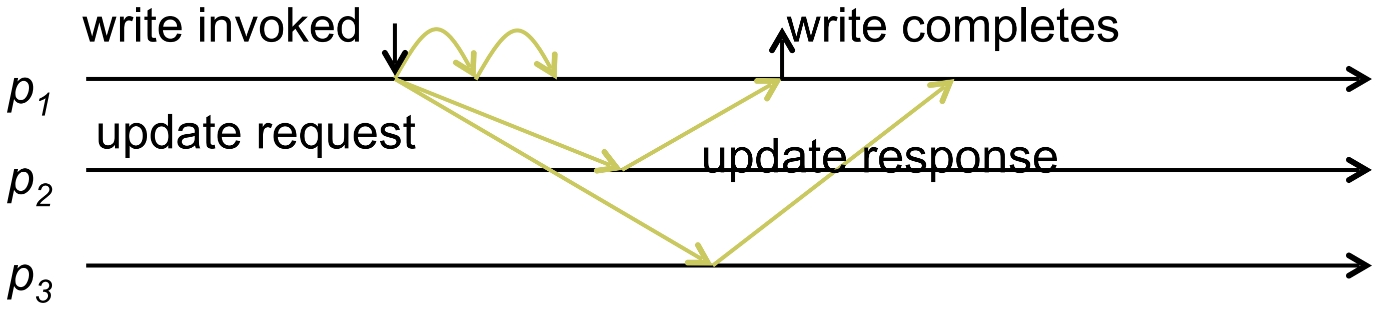
Phases

- The communication involved in operations are structured 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



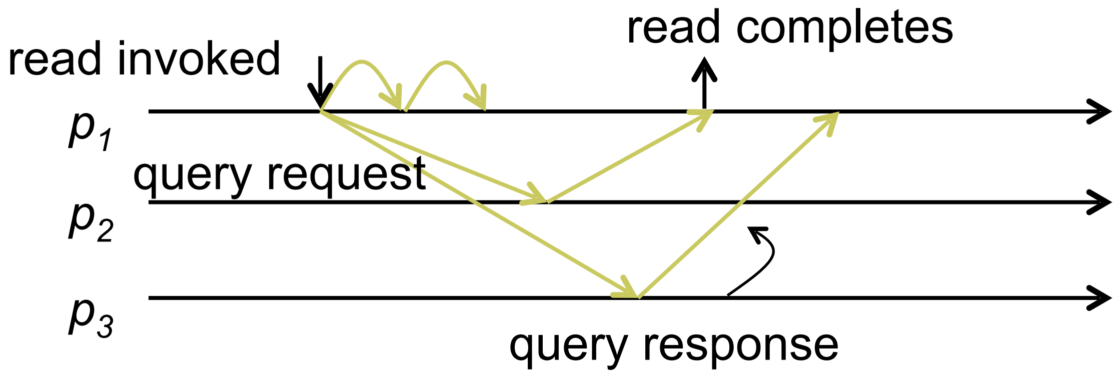
Write operation Majority Voting

- 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 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 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 (ts, v)$_j$
  - When query phase ends, p_i picks **max (ts, v)$_j$ received**



Majority Voting Algorithm (1,N)

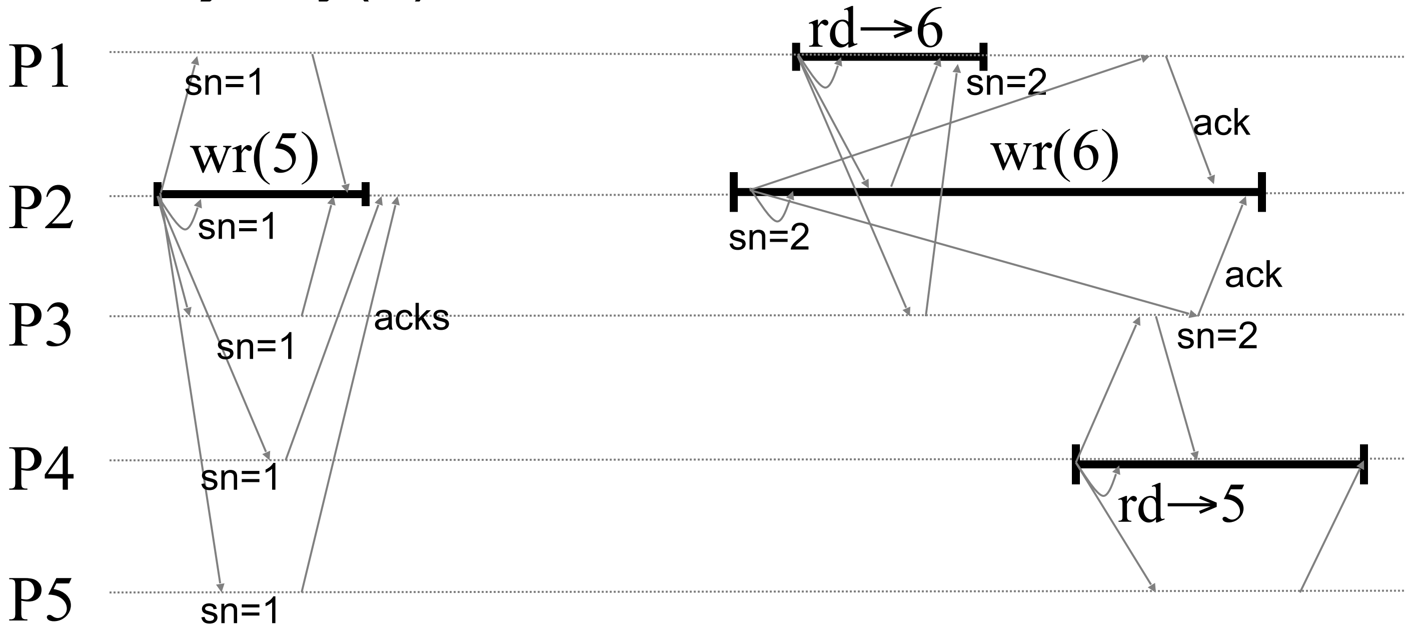
- Assume majority of ***correct processes***
  - Register values have a sequence number (seq#)
  - No FD
- to **write(v)**
  - **ts**++
  - Broadcast **v** and **ts** to all
    - if newer **ts**:
      - Receiver update to (**ts, v**)
    - Receiver sends ACK
  - Wait for ACK from ***majority of nodes***
  - Return
- 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 **update phase** with (v,ts)

The read **query phase**

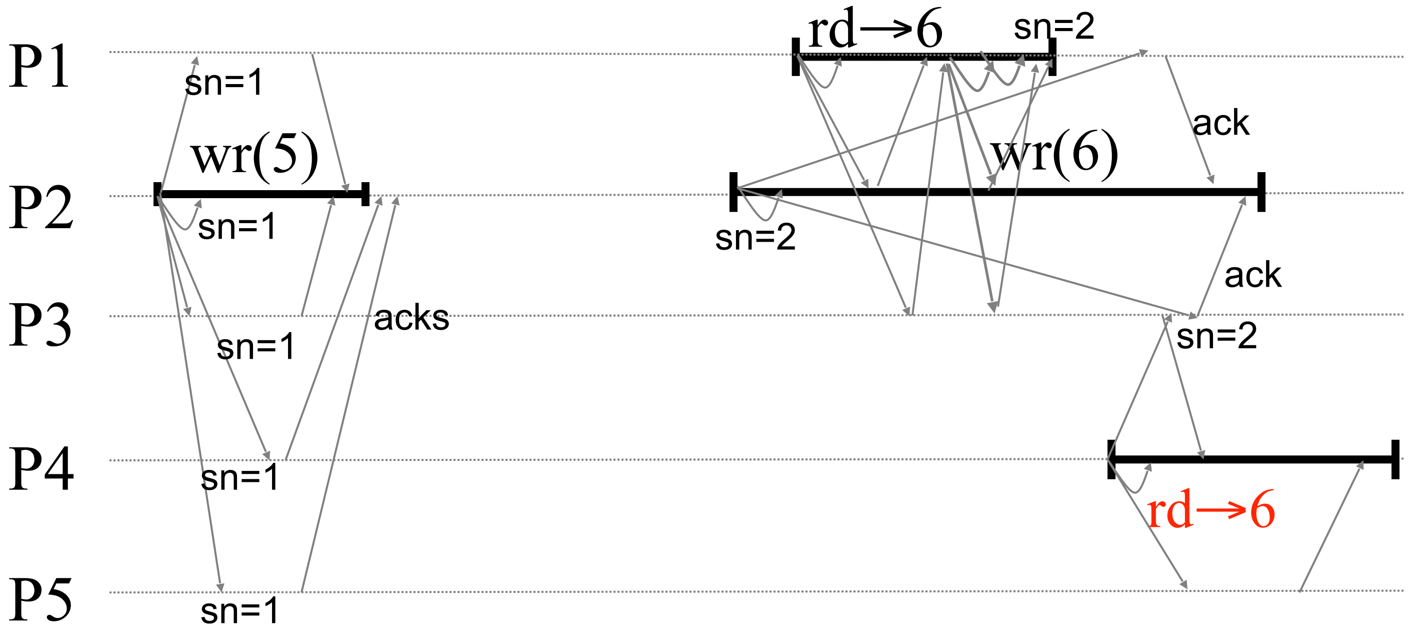
Regular but not Atomic

- Problem with majority voting
- Ex: majority(5)=3



Regular but not Atomic

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

    ← query phase
  - 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

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

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

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"

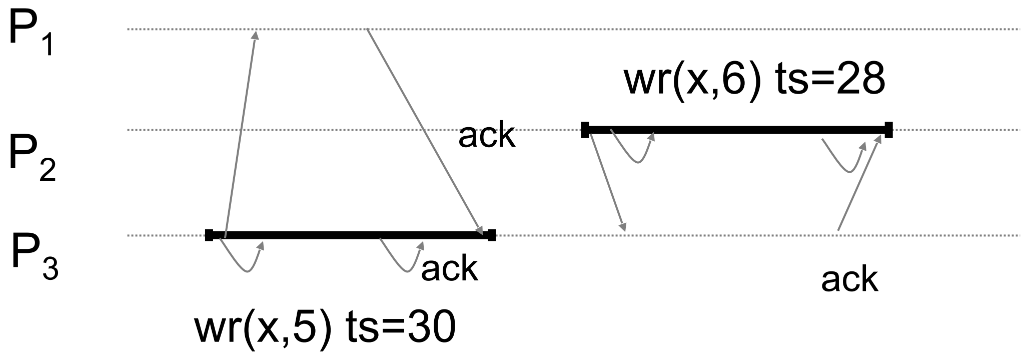
Ordering

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

(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



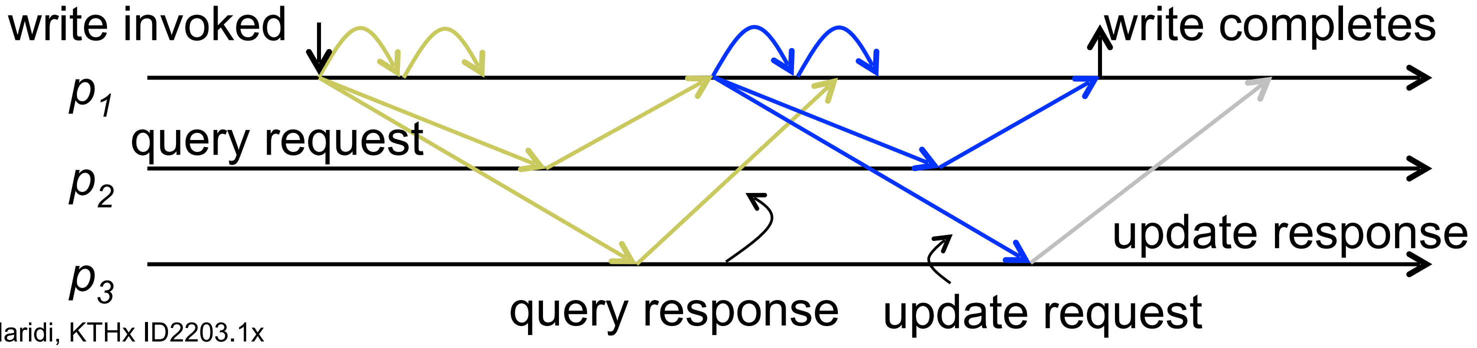
Atomic Registers (N,N) 1/2

- ***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? [d]
  - 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 phase, any process p_i sends ACK message even if pi receive 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 (query phase)
  - A read to complete requires 2 round-trips of messages is
    - One for read (query phase)
    - One for impose if necessary (query 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)

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($T|x_r$) 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

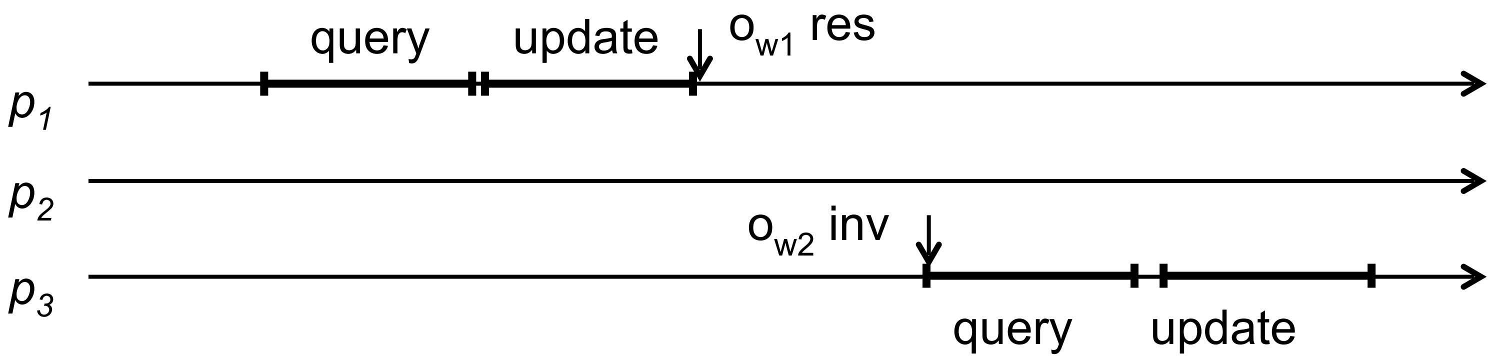
- ✓ S preserves non-overlapping order as $T|x_r$
- S and $T|x_r$ are equivalent
  - They contain same events
  - $(T|x_r)|p_i$ contains non-overlapping operations
  - $(T|x_r)|p_i = S|p_i$
- Hence, LIN($T|x_r$) for any register x_r, which implies LIN(T)

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 $o_1 <_S o_2$
  - $res(o_1) <_{T'} inv(o_2) \Rightarrow res(o_1) <_s inv(o_2)$

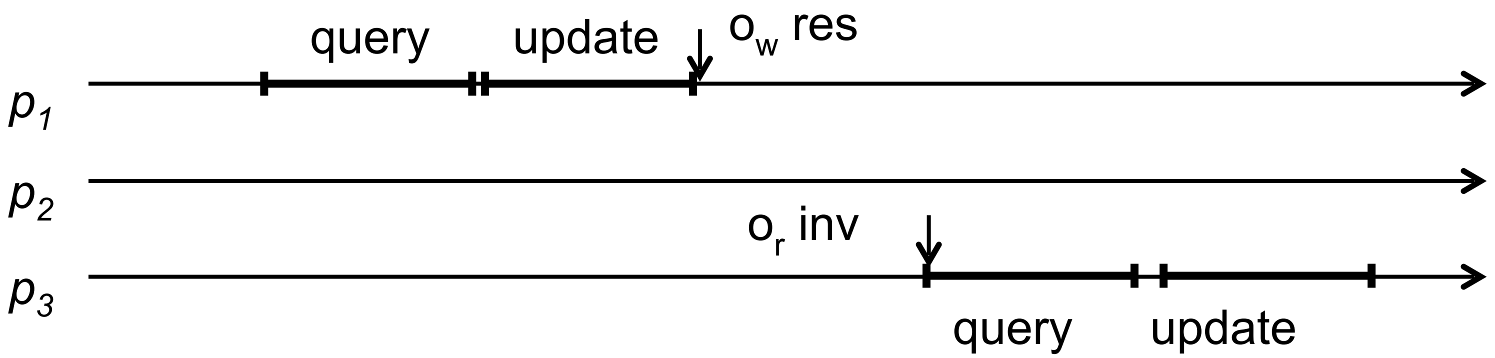
O1 and O2 are write operations

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



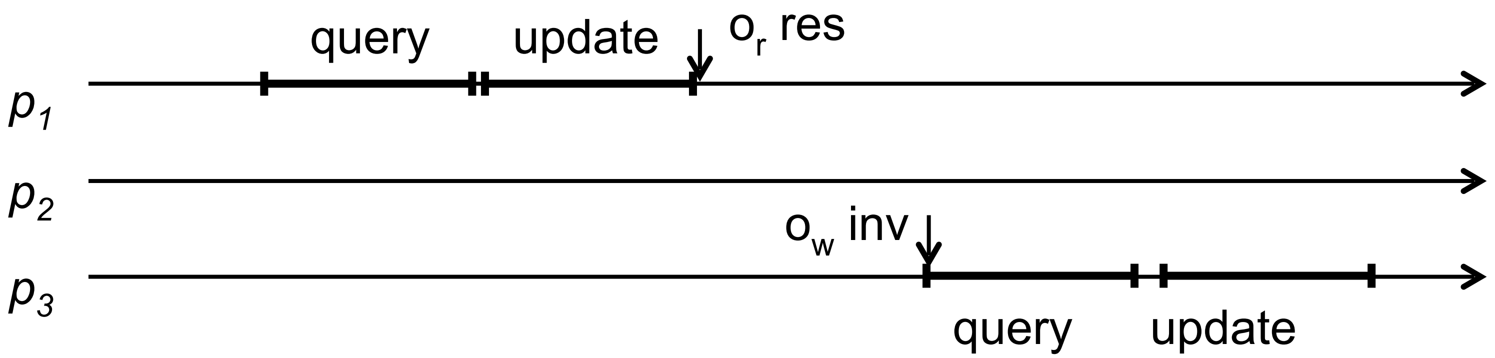
O1 (o_w) write and O2 (o_r) is read

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



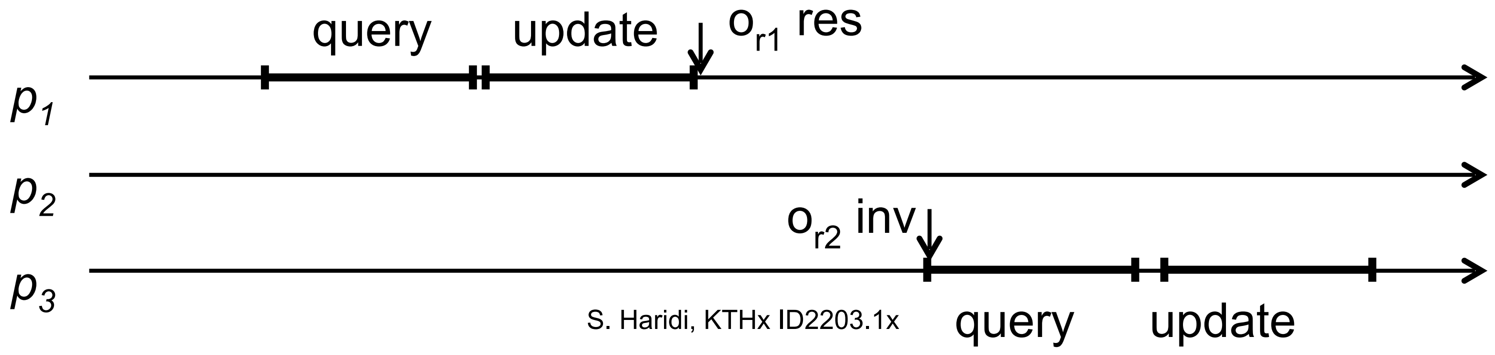
o_1 (o_r) is read and o2 (o_w) is write

- $o_r <_{H'} o_w \Rightarrow o_r <_s o_w$
- $res(o_r) <_{H'} inv(o_w) \Rightarrow \mathrm{ts}(o_r) < \mathrm{ts}(o_w)$
- 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 $\mathrm{ts}(o_r) < \mathrm{ts}(o_w) \Rightarrow \mathrm{ts}(o_r) < \mathrm{ts}(o_w)$



o_1 (o_{r1}) is read and o2 (o_{r2}) is read

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



(N,N) Algorithm for Sequentially Consistent Registers

Sequentially Consistent Algorithm LT

- In Fail-silent model implement read/write **multiple register shared memory**
  - Multiple writers and multiple readers
  - Sequentially consistent model (SC)
  - Writes in 1 RTT and reads in 2 RTTs*
  - Tolerates $f < n/2$ faulty processes failing by crashing

- Logical Time **(LT) algorithm**

Compositionality of the SC algorithm

- Unlike linearizability (LIN), sequential consistency (SC) is not a compositional consistency condition

- LT-algorithm satisfies SC, but also satisfies a bit stronger consistency condition that is compositional
  - Simple correctness proof
  - Scalability: allows for storing subsets of registers in different replication groups, a.k.a. *sharding*

Compositionality of LT-algorithm

- LT-algorithm linearizes reads/writes in logical time instead of real-time
- Executions (real-time traces) are sequentially consistent instead of linearizable
- Linearizability in logical time allows compositionality
- LT-algorithm satisfies SC, but also satisfies a bit stronger consistency condition that is compositional

THE LT ALGORITHM

Logical time and clocks

- The happens-before relation $e \rightarrow e'$:
  - If e occurs before e' in the same process
  - If e is the sending of msg m and e' is the receipt of m
  - If exists e'' such that $e \rightarrow e''$ and $e'' \rightarrow e'$

- Logical clocks assign to each e a logical time $lt(e)$
  - s.t. $e \rightarrow e' \Rightarrow lt(e) < lt(e')$

- LT-algorithm implements logical clock, one per process, as an integer variable named lt that is updated at beginning of each step as follows:
  - Local (invocation) step: $lt := lt+1$
  - Send message m: include current logical time as $m.lt$
  - Receive message m: $lt := \max(lt, m.lt)+1$

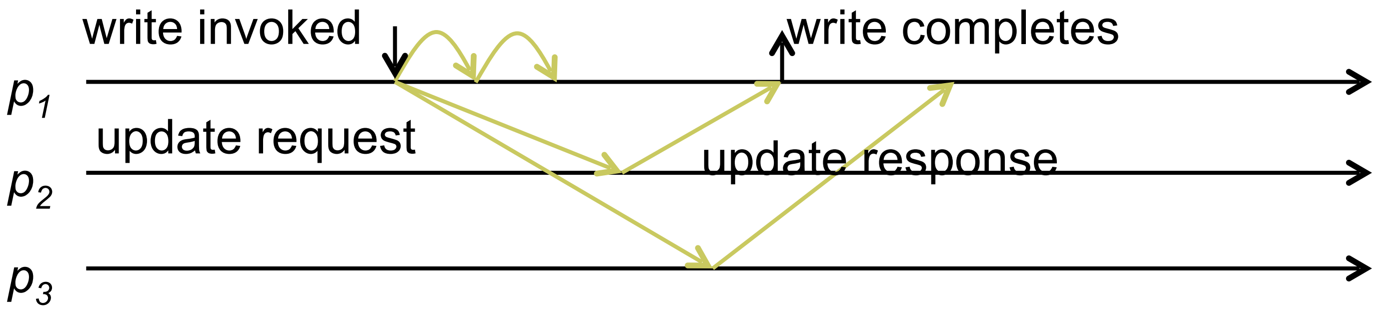
Timestamp-value pairs

- Each process stores the values of all registers
- Value of register r
  - is timestamp-value pair, *tvp*=(*ts*, *v*)
  - *ts* determine which value is more recent
- *ts* is a *logical time-pid pair, ts*=(*lt*, *i*) for process p_i

- Each process pi
  - Stores register value with max timestamp for each register r
  - Initially ((0, i), ⊥) for all registers r

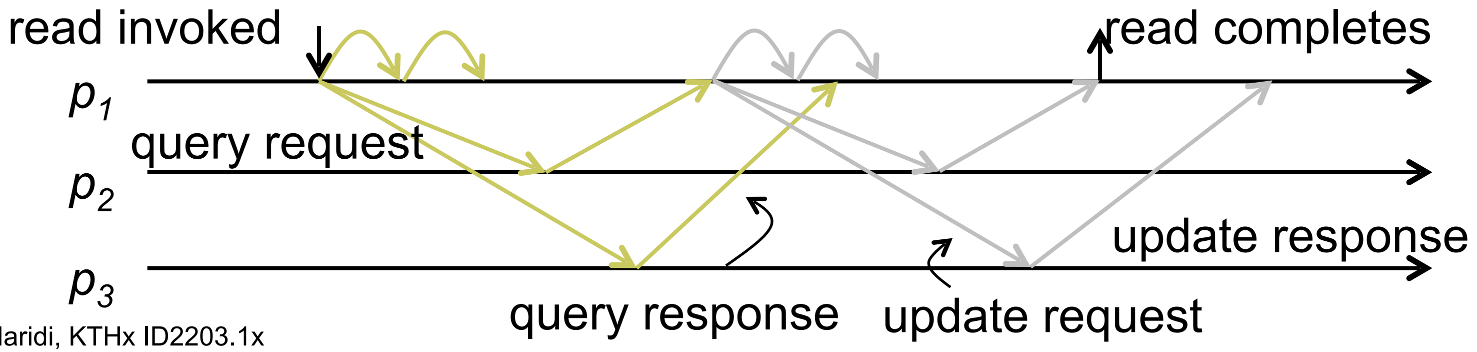
Write operation

- Process p_i executing write(r, v) operation
  - p_i forms $tvp = ((lt, i), v)$, where lt is current logical time
  - p_i starts an *update phase* by sending update request with register id r and timestamp-value pair tvp
  - p_j updates the value of r to max(r, tvp) and responds
  - p_i completes write when update phase ends



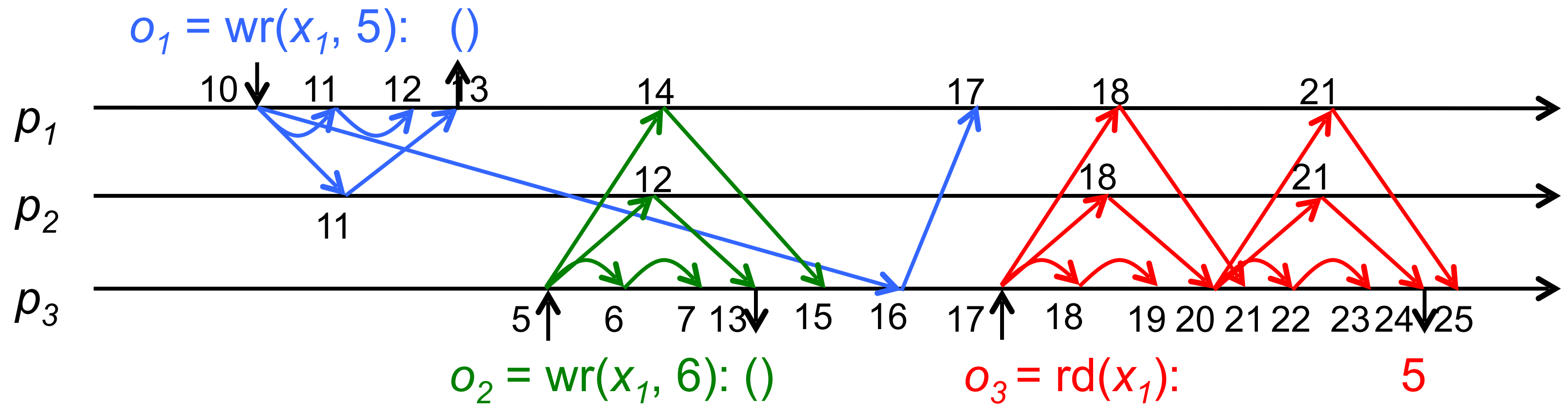
Read operation

- 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 current tvp_r for register X_r
  - When query phase ends, p_i picks max tvp received
  - p_i does an update phase with id r and tvp
  - After update phase ends, p_i returns with value v of tv



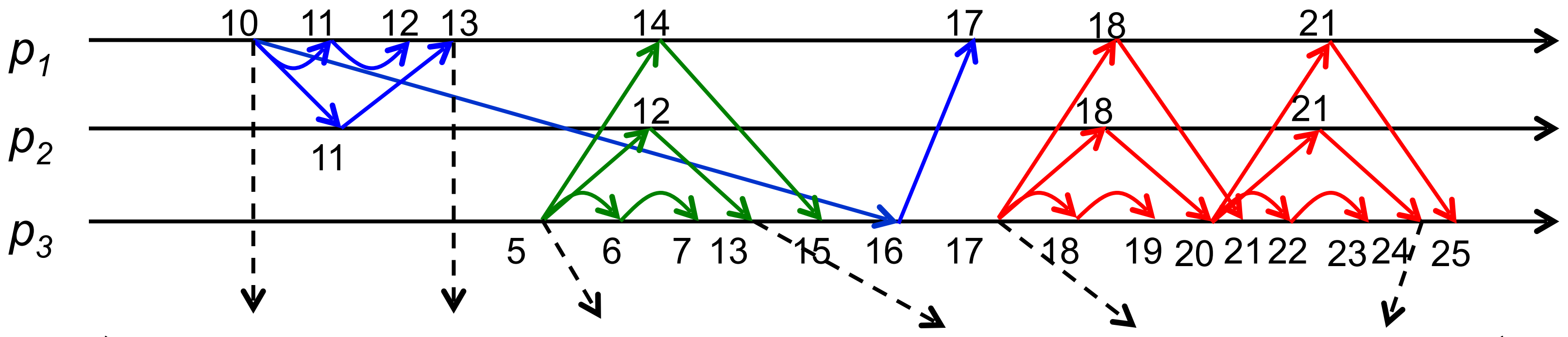
Example

- Show each system execution is sequentially consistent
- Running example
  - Three operations
  - Numbers are logical times for each step (used later)



Trace T

- Reason about sequence of interface events (invocations & responses) in execution, ordered by real time



$T = \langle$winv(p_1,x_1,5), wres(p_1), winv(p_3,x_1,6), wres(p_3), rinv(p_3,x_1), rres(p_3,5)$\rangle$

Sequential Consistency (SC)

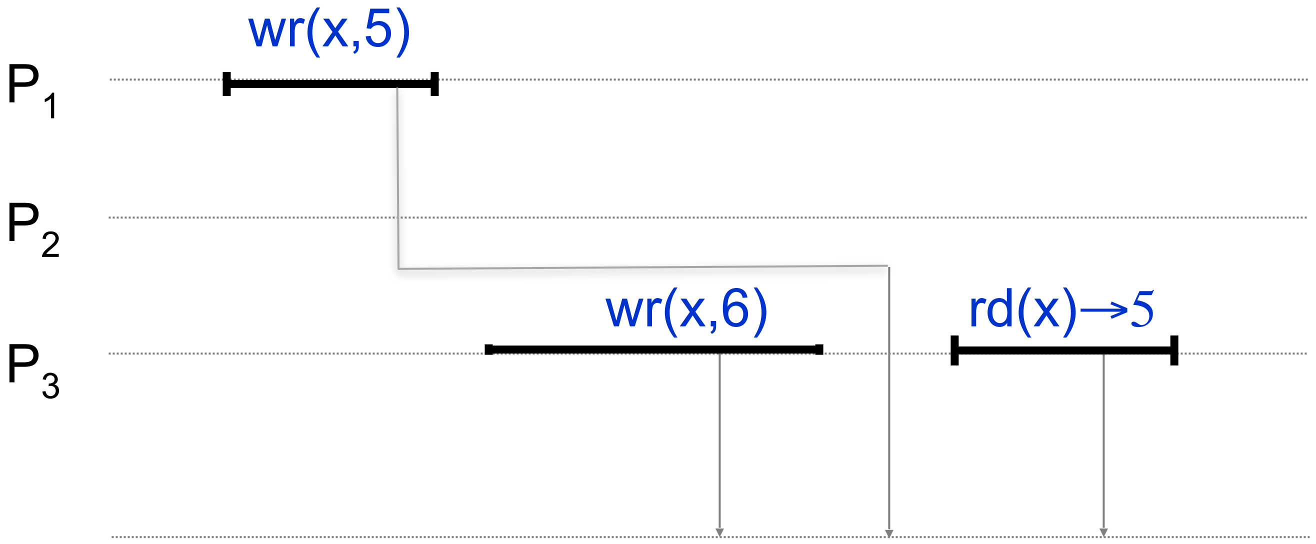
- *T* is legal
  - *T* is sequential (no overlapping ops) and each read returns last value written to register

- Traces *S* and *H* are equivalent written as $S \simeq H$
  - if $\forall p_i$: $S|p_i = H|p_i$

- SC as property on history *T*:
  - SC(*T*) if there exists legal trace *S* such that $S \simeq H$

Sequential Consistency (SC)

- $H = \langle \text{winv}(p_1,x_1,5), \text{wres}(p_1), \text{winv}(p_3,x_1,6), \text{wres}(p_3), \text{rinv}(p_3,x_1), \text{rres}(p_3,5) \rangle$
- $S = \langle \text{winv}(p_3,x_1,6), \text{wres}(p_3), \text{winv}(p_1,x_1,5), \text{wres}(p_1), \text{rinv}(p_3,x_1), \text{rres}(p_3,5) \rangle$



CORRECTNESS OF LT ALGORITHM

Linearizability (LIN)

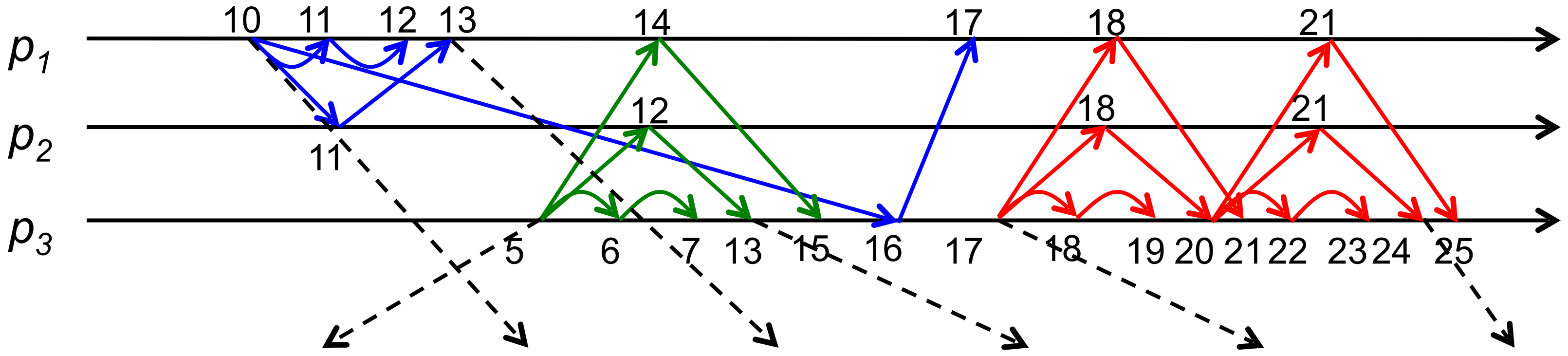
- LIN is a consistency condition similar to SC
  - LIN(T) requires that there exists legal history S:
    - S is equivalent to T,
    - **If $o_1 <_T o_2$ then it must also be that $o_1 <_S o_2$**

- LIN is stronger than SC: LIN(T) $\Rightarrow$ SC(T)

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

Logical time trace T^{lt}

- Reorder events according to logical times when they occurred, breaking ties using process identifiers



$H^{lt} = \langle \text{winv}(p_3,x_1,6)_5, \text{winv}(p_1,x_1,5)_{10}, \text{wres}(p_1)_{13,1}, \text{wres}(p_3)_{13,3}, \text{rinv}(p_3,x_1)_{18}, \text{rres}(p_3,5)_{24} \rangle$

- Note that o_1 and o_2 are concurrent in T^{lt}, but not in T
- LIN(T^{lt}) holds:
  - $S = \langle \text{winv}(p_3,x_1,6), \text{wres}(p_3), \text{winv}(p_1,x_1,5), \text{wres}(p_1), \text{rinv}(p_3,x_1), \text{rres}(p_3,5) \rangle$

Properties of T^{lt}

- For any two events e, e' in $T|p_i$:
  - The order between them is maintained in $T^{lt}|p_i$
    - Because logical time preserves process order
- Therefore, $T \simeq T^{lt}$
- And $\mathrm{SC}(T) \Leftrightarrow \mathrm{SC}(T^{lt})$
  - As there exists a legal history S s.t. $S \simeq H \simeq H^{lt}$
- Taken together with def. of LIN, we have:
  - $(\forall x_r\colon \mathrm{LIN}(T^{lt}|x_r)) \Rightarrow \mathrm{LIN}(T^{lt}) \Rightarrow \mathrm{SC}(T^{lt}) \Rightarrow \mathrm{SC}(T)$
- We can reason compositionally!

Completing the proof

- We must show that, for each execution, and for each register x_r, LIN($H^{lt}|x_r$) holds

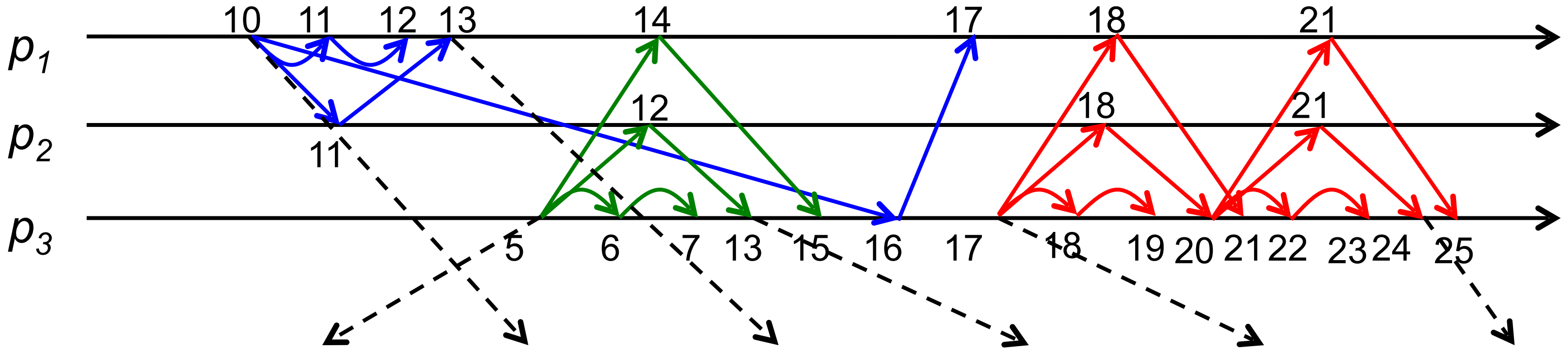
  - Requires that there exists legal history S s.t.
    - S is equivalent to $H^{lt}|x_r$,
    - S preserves order of non-overlapping ops in $H^{lt}|x_r$

  - Done by constructing S and showing it has the required properties…

Timestamp order

- Timestamp of operation *o*, *ts*(*o*), is timestamp used in *o*'s update phase
- Construct *S* from $H^{lt}|x_r$ in timestamp order:
  1. Order writes according to their (unique) ts
  2. Order each read immediately after write with same time stamp
     - For reads with same ts, order them by increasing logical time of invocation step
- *S* is legal by construction

S of history H^{lt} for register *x*



$H^{lt} = \langle winv(p_3,x_1,6)_5,\ winv(p_1,x_1,5)_{10},\ wres(p_1)_{13,1},\ wres(p_3)_{13,3},\ rinv(p_3,x_1)_{18},\ rres(p_3,5)_{24} \rangle$

- $ts(write(x,5)_{p1}) = (10,1)$, $ts(write(x,6)_{p3}) = 5$, $ts(read(x)_{p3}) = (10,1)$

$S = \langle win(p_3,x_1,6),\ wr(p_3),\ win(p_1,x_1,5),\ wr(p_1),\ rin(p_3,x_1),\ rr(p_3,5) \rangle$

Completing the proof

- We must show that, for each execution, and for each register x_r, LIN($T^{l/t}|x_r$) holds

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

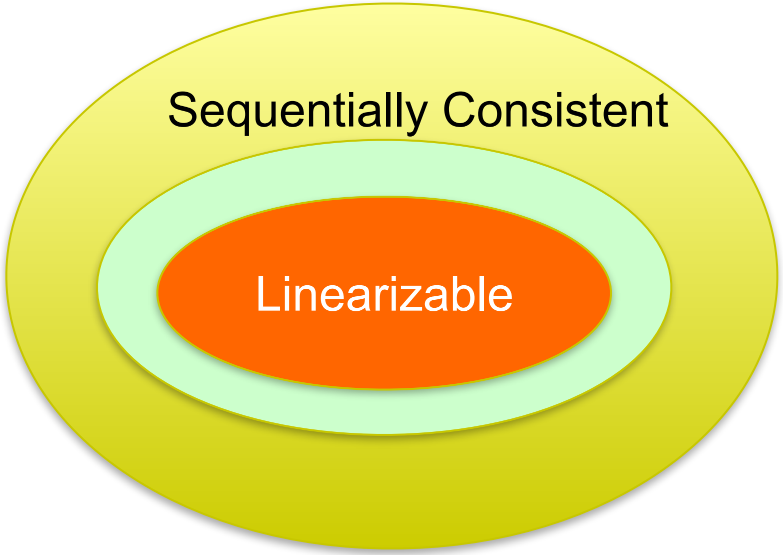
Equivalence

- ✓ S preserves non-overlapping order as $H^{lt}|x_r$
- S and $H^{lt}|x_r$ are equivalent
  - They contain same events
  - $(H^{lt}|x_r)|p_i$ contains non-overlapping operations
  - $(H^{lt}|x_r)|p_i = S|p_i$

- Hence, LIN($H^{lt}|x_r$) for any register x_r, which implies SC(H)

The End

- Consistency condition on histories (H)
- SC(H) and for each reg. x
- $o_{w(x)}$ is write and $o_{r(x)}$ is read, $o_{w(x)} <_H o_{r(x)} \Rightarrow o_{w(x)} <_S o_{r(x)}$
- $o_{r(x)} <_H o_{w(x)} \Rightarrow res(o_{r(x)}) \neq val(o_{w(x)})$



Relation to other algorithms

	Single writer	Multi writers	Multi writers
Writers	Single writer	Multiple writers	Multiple writers
Consistency	Linearizability	Linearizability	Sequential consistency
Time complexity	Write: 1 RTT Read: 2(*) RTTs	Write: 2 RTTs Read: 2(*) RTTs	Write: 1 RTT Read: 2(*) RTTs
Fault-tolerance	$f < n/2$	$f < n/2$	$f < n/2$

(*) For all of these algorithms, the 2nd RTT in a read can often be skipped (if there are no current writes)

Preserving non-overlapping order 1/2

- Must show that S preserves the order of non-overlapping ops in $H^{lt}|x_r = H'$
  - $res(o_1) <_{H'} inv(o_2) \Rightarrow res(o_1) <_s inv(o_2)$
- By definition of logical time history:
  - $res(o_1) <_{H'} inv(o_2) \Rightarrow lt(res(o_1)) \leq lt(inv(o_2))$
- If o_1 has update phase and o_2 has query phase, and $lt(res(o_1)) \leq lt(inv(o_2))$, then $ts(o_1) \leq ts(o_2)$
  - Process p_k responds in both phases, and processes update request before query request

Preserving non-overlapping order

- Case analysis:
  - o_1 and o_2 are writes: $ts(o_1).lt = lt(inv(o_1)) < lt(res(o_1)) \leq lt(inv(o_2)) = ts(o_2).lt \Rightarrow o_1 <_S o_2$
  - o_1 read, o_2 write: there exists a write o_0 such that $ts(o_0)=ts(o_1)$ and $lt(inv(o_0))<lt(res(o_1))$ by causality
    - $ts(o_1).\text{lt} = lt(inv(o_0)) < lt(res(o_1)) \leq lt(inv(o_2)) = ts(o_2).lt$
  - o_1 read/write, o_2 read: from last slide, $ts(o_1)\leq ts(o_2)$
    - If read-read and $ts(o_1)=ts(o_2)$ then $o_1 <_S o_2$ as $lt(inv(o_1)) < lt(inv(o_2))$

Simulating Message Passing?

- So asynchronous model can simulate shared Read/Write shared memory
  - Majority of correct nodes is all that is needed
- So if a problem P is solvable in Read/Write shared memory, it is also solvable in the asynchronous model

Simulating Message Passing?

- Can we simulate the asynchronous model in Read/Write shared memory? [d]
  - Yes. One register AB for every channel (unbounded value)
    - Modeling a directed channel from A to B
    - Each sender and receiver A keeps track of this register locally in ab
  - Sending msgs by appending to right channel
    - WRITE(AB, ab + [m])
  - Receive msgs by busy-polling incoming “channels”
- Shared memory and message passing equivalent

Equivalence of SM and Asynchronous Model

- So Read/Write Shared memory and the asynchronous model are equivalent
- So since solving consensus is impossible in the asynchronous model, it also impossible in Read/Write Shared memory

Summary

- Shared Memory registers for read/write
  - Consistency of data in the presence of failures and concurrency

- Regular Register (the weak model)
  - Bogus algorithm (didn't work)
  - Centralized algorithm (no failures)
  - Read-One Write-All Algorithm (Perfect FD)
  - Majority Voting (No FD)

- Atomic Register (the strong model)
  - Single writers
    - Read-Impose Idea (make sure reads are ordered)

  - Multiple Writers
    - Read-Impose Write-consult-majority
    - Before write, get highest sequence number