Shared Memory

Seif Haridi – Royal Institute of Technology
Peter Van Roy – Université catholique de Louvain

haridi(at)kth.se
peter.vanroy(at)uclouvain.be

Real Shared Memory

- Formal model of shared memory
  - No message passing (no outbuf/inbuf, del)
  - Instead all nodes access one shared memory
  - Models multiprocessors, multicores…
- We are interested in distributed systems
  - Simulate shared memory using message passing

Simulating Shared Memory

- "Simulate" that the DS has ***shared memory***
  - A ***register*** represents each memory location
    - Registers aka objects
  - Nodes can ***read*** / ***write*** to registers
    - Not only RW-registers… FIFO-queue…
  - This is a simplification of key-value stores
    - Practical in many large Web 2.0 applications

Slides by Seif Haridi and Ali Ghodsi

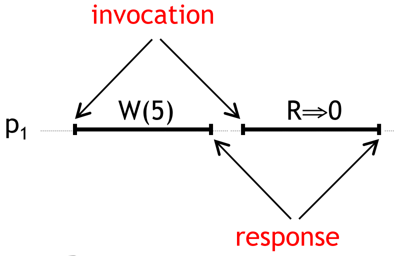
Why simulate shared memory?

- Why?
  - We're really just studying consistent replication (ultimately single system illusion)
  - Replicate for fault tolerance and scalability
- Challenges
  - Provide consistency in presence of failures
  - Provide consistency in presence of concurrency

Slides by Seif Haridi and Ali Ghodsi

Read/Write Register

- RW-registers have 2 operations
  - read(R)⇒x
    - Value of R was read to be x
  - write(R, x)
    - Update register R to value x
  - Sometimes omit register name R
    - Specification with respect to one register



Basic Assumptions

- Nodes are sequential
  - invocation, response, invocation, response,…
  - I.e. do one operation at a time
- Register values (simplifying assumption)
  - Values are positive integers, initially zero

Definitions

- In an execution, an operation is
  - complete if both invocation & response occurred
  - failed if invoked, but no response arrives
- op_1 precedes op_2 if (denoted $<_p$)
  - Response of op_1 precedes invocation of op_2
- op_1 and op_2 are concurrent if neither precedes the other

Terminology

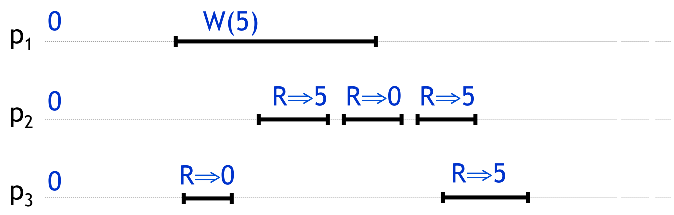
- (1,N)-algorithm
  - 1 designated writer, multiple readers
- (N,N)-algorithm
  - Multiple writers, multiple readers

Regular Registers

Regular Register (1, N)

- Termination
  - Each read and write operation of a correct node completes
- Validity
  - Read returns ***last value written*** if
    - Read is not ***concurrent*** with another write, and
    - Read is not concurrent with a ***failed operation***
  - Otherwise the read must return the last value written or a concurrent value being written

Example



- Regular? yes
  - Not a single storage illusion!

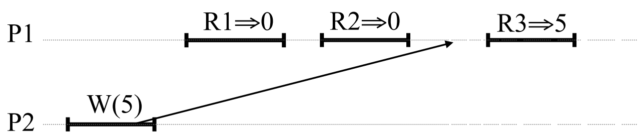
Centralized Algorithm

- 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

Slides by Seif Haridi and Ali Ghodsi

Bogus Algorithm (regular)

- Intuitively: make an algorithm in which
  - A read just reads local value
  - A write writes to all nodes
- to **write(v)**
  - Update local value to v
  - Broadcast v to all (each node locally updates)
  - Return
- to **read**
  - Return local value
- ***Problem?*** [d]

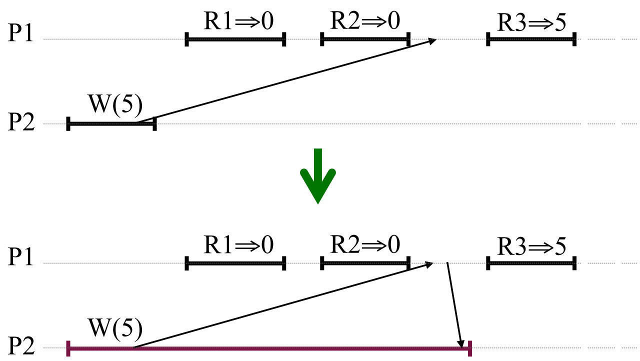


Read-one Write-All (1,N)

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

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

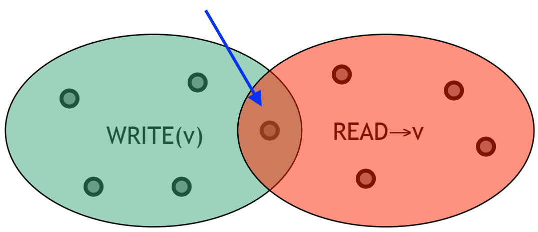
- ***Main idea***
  - Postpone write responses



Majority Voting Algorithm

Fail-Silent model

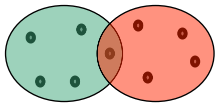
- ***Main idea***
  - Quorum principle (for example: majority)
    - Always write to and read from a majority of nodes
    - At least one node knows most recent value



    - 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



- Maekawa Quorum
  - Arrange nodes in a MxM grid (M=sqrt(N))
  - Write to rows, read to columns (always overlap)
  - **Pro**: Only need to read/write sqrt(N) nodes
  - **Con**: Tolerate at most sqrt(N)-1 crashes

P1	*P2*	*P3*
P4	*P5*	*P6*
P7	*P8*	*P9*

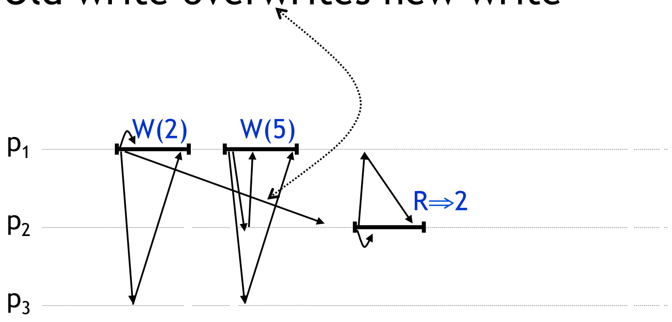
- What about other quorums?

Majority Voting Algorithm (1,N)

- Assume majority of ***correct processes***
  - Register values have a sequence number (seq#)
  - No FD
- to **write(v)**
  - Broadcast v and seq# to all
    - Receiver update to v
  - Wait for ACK from ***majority of nodes***
  - seq#++
  - Return
- to **read**
  - Broadcast read request to all
    - Receiver respond with local value and seq#
  - Wait and save values from ***majority of nodes***
  - Return value with ***highest*** seq#

Problem with algorithm

- Old write overwrites new write



Majority Voting Algorithm (1,N)

- Assume majority of ***correct processes***
  - Register values have a sequence number (seq#)
  - No FD
- to **write(v)**
  - Broadcast v and seq# to all
    - Receiver update to v if ***newer*** seq#
  - Wait for ACK from ***majority of nodes***
  - Return
- to **read**
  - Broadcast read request to all
    - Receiver respond with local value and seq#
  - Wait and save values from ***majority of nodes***
  - Return value with ***highest*** seq#

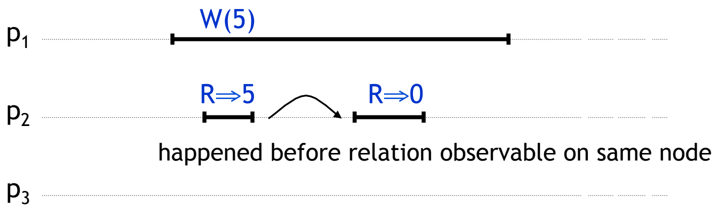
Towards single storage illusion...

Safety: consistency

- 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

Motivating Example 1

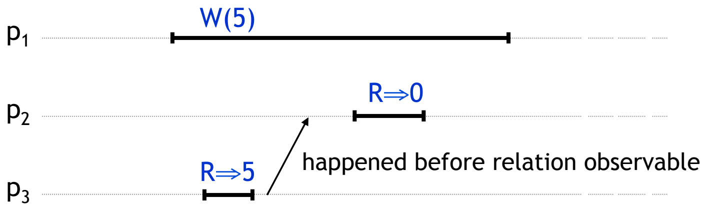
- Regular execution



- Sequential consistency disallows such E's

Motivating Example 2

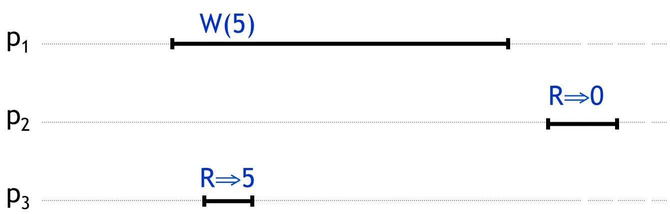
- Regular execution



- Sequential consistency allows such E's

Motivating Example 2

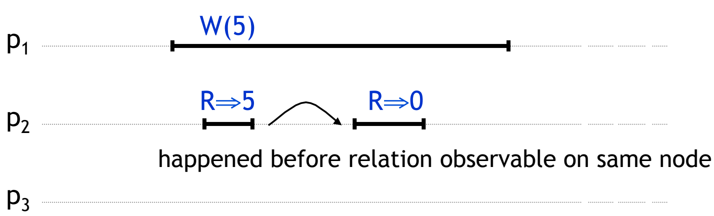
- Sequentially consistent execution



- Regular consistency disallows such an execution

Motivating Example 1

- Regular execution

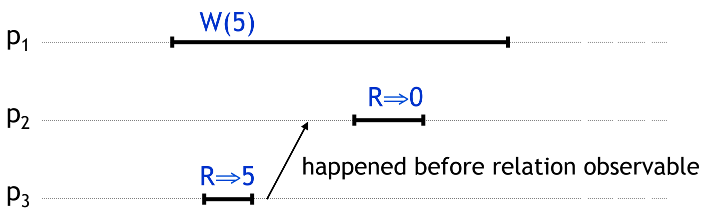


- Atomicity/Linearizability disallows such E’s
  - No single storage could behave that way

Slides by Seif Haridi and Ali Ghodsi

Motivating Example 2

- Regular execution



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

Liveness: progress

- Liveness requirements: three progressively weaker versions
  - Wait-free (strongest)
    - Informally:
      Every correct node should “make progress”
      (no deadlocks, no livelocks, no starvation)
  - Lock-free/non-blocking
    - Informally:
      At least one correct node should “make progress”
      (no deadlocks, no livelocks, maybe starvation)
  - Obstruction free/solo-termination (weakest)
    - Informally:
      if a single node executes without interference (contention) it makes progress
      (no deadlocks, maybe livelocks, maybe starvation)

Atomic/Linearizable Registers

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

Alternative Definition

Linearization points

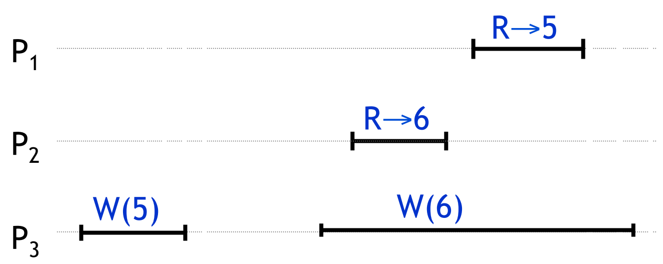
- Read ops appear as immediately happened at all nodes at
  - some time between invocation and response
- Write ops appear as immediately happened at all nodes at
  - some time between invocation and response
- Failed ops appear as
  - completed at every node XOR
  - never happened at any node

Ordering (only for (1,N))

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

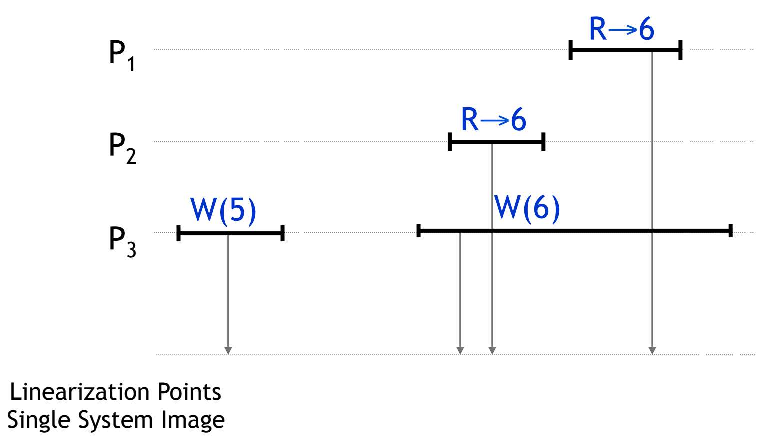
same

Example

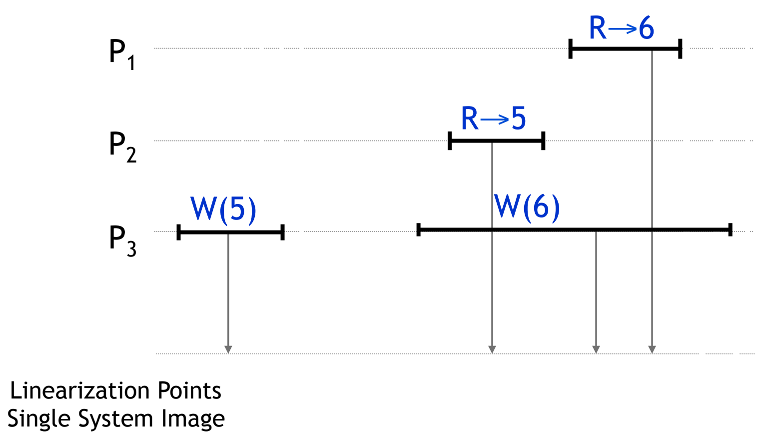


- Atomic? [d]
  - No, not possible to find linearization points
  - W(5)<R(5)<W(6)<R(6) ⇒ R(5)<R(6)
    - But R(6) < R(5)

Example 2

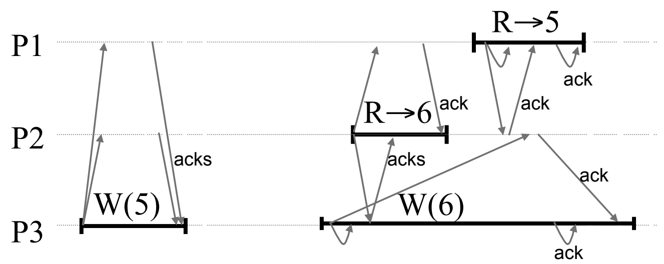


Example 2



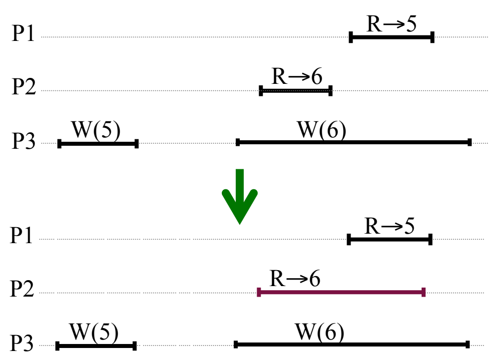
Regular but not Atomic

- Problem with majority voting
- Ex: majority(3)=2



Atomic Registers (one writer)

- ***Main idea***
  - **Read-impose**
    - When reading, also make a write before responding



Slides by Seif Haridi and Ali Ghodsi

Why does it work?

- A read R makes a write
  - Any read after R must at least see R
- ***Causality used to enforce atomicity***

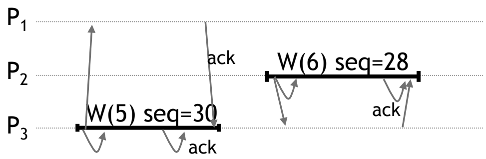
- Validity
  - Read returns ***last value written*** if
    - Read not ***concurrent*** with another write
    - Read not concurrent with a ***failed operation***
  - Otherwise read must return last or concurrent value being written
- Ordering
  - If a **read→r1** precedes **read→r2**
  - Then **write(r1)** precedes **write(r2)**

Atomic Registers (one writer)

- ***Main idea***
  - **Read-impose**
    - When reading, also make a write before responding
  - **Optimization [d]**
    - When reading, if majority have same seq # do not write

Atomic Register (multiple writers)

- Read-Impose Majority Voting
  - Multiple writers might have non-synchronized sequence numbers
- Example:
  - The latter W(6) is ignored because old seq#



Atomic Registers (N,N) 1/2

- ***Main idea in read-impose write-consult-majority (N,N)***
  - **Get Seq#**
    - Before writing,
      - read from majority to get last seq#
- Problem
  - Two concurrent writes with same sequence#? [d]
  - Just compare process id, break ties!

Atomic Registers (N,N) 2/2

- ***Main idea in read-impose write-consult-majority (N,N)***
  - **Get Seq#**
    - Before writing,
      - read from majority to get last seq#
- Send ACK if receive write with old seq#? [d]
  - Yes, because of multiple writers
  - Example:
    - Slow P1 writes(5), waits for acks: needs acks to complete
    - Fast P2 writes(6), receives acks from majority
    - P1 does not get enough acks, as nodes ignore its write(5)

Simulating Message Passing?

- So we can simulate shared memory
  - Majority of correct nodes is all that is needed
- Can we simulate message passing in shared memory? [d]
  - Yes. One register AB for every channel
    - Modeling a directed channel from A to B
  - Send msgs by appending to the channel
  - Receive msgs by busy-polling incoming “channels”
- Shared memory and message passing equivalent
  - In functionality, not always in efficiency!

Summary

- Shared Memory registers for read/write
  - Consistency of data in the presence of failures and concurrency
- Regular Register (the weak model) (1,N)
  - Bogus algorithm (didn't work)
  - Centralized Algorithm (no failures)
  - Read-One Write-All Algorithm (Perfect FD)
  - Majority Vote or Quorum (No FD)
- Atomic Register (the strong model)
  - Single Writers (1,N)
    - Read-Impose Algorithm (makes sure reads are ordered)
  - Multiple Writers (N,N)
    - Read-Impose Write-Consult-Majority Algorithm
    - Before write, get highest sequence number (makes sure seq nos. are synched)

Linearizability Formally

- Every execution consists of
  - R-inv$_i$(X)
    - Read invocation by node i on register X
  - R-res$_i$(a)
    - Response with value a to read by node i
  - W-inv$_i$(X,a)
    - Write invocation by node i on register X with value a
  - W-res$_i$
    - Response (confirmation) to write by node i

Executions formally

- Every execution consists of
  - Read operations which consist of two events
    - $R\text{-}inv_i(X)$
      - Read invocation by node i on register X
    - $R\text{-}res_i(a)$
      - Response with value a to read by node i
  - Write operations which consist of two events
    - $W\text{-}inv_i(X,a)$
      - Write invocation by node i on register X with value a
    - $W\text{-}res_i$
      - Response (confirmation) to write by node i
- An execution 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, …

Basic Assumptions

- Assumptions on executions (well-formed)
  - First event of every node is an invocation
  - A node alternates between invocations and responses
- An operation O is pending in execution E if
  - O has no response event
  - Otherwise O is complete
- An execution is complete if
  - Every operation is complete
  - Otherwise it is partial
- An operation X precedes operation Y in exec E
  - If response of X is before invocation of Y in E

Linearizability Formally (no failure)

- A complete execution E is linearizable if there exists an execution F s.t.
  - Similarity
    - E and F contain same events
  - No concurrency
    - F is sequential
  - Legal operations
    - Read resp have value of preceding write inv in F
  - Time ordering
    - If op X precedes Y in E, then X precedes Y in F

Linearizability Formally (failure)

- No observable failures in complete executions
- Linearizability for partial executions (failures)
  - A partial execution E is linearizable if E is modified to F s.t.
    - Every pending operation is completed by
      - Removing the invocation of the operation, or
      - Adding a response to the operation
    - F is linearizable

Sequential Consistency Formally

- Event X locally precedes Y in E if
  - X and Y occur at same node and X precedes Y in E

- An execution E is sequentially consistent if there exists an execution F s.t.
  - Similarity
    - E and F contain same events
  - No concurrency
    - F is sequential
  - Legal operations
    - Read resp have value of preceding write inv in F
  - Local time ordering
    - If op X locally precedes Y in E, then X locally precedes Y in F