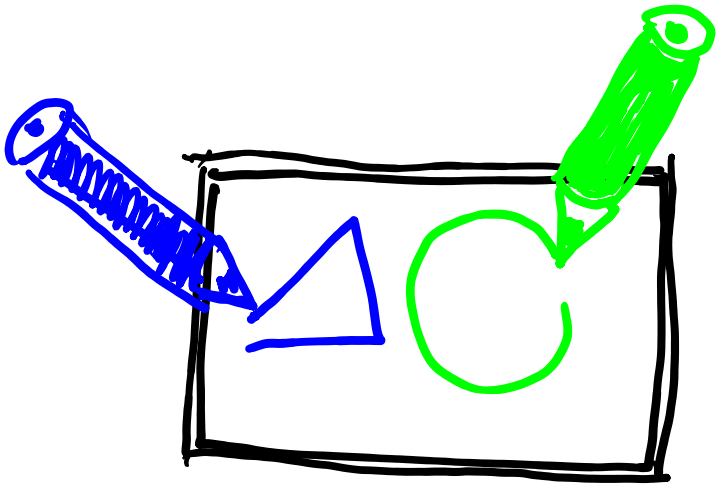
Storage
in message passing systems



SLR206, P2

So far…

Shared-memory computing:

- Wait-freedom and linearizability
- Lock-based and lock-free synchronization
- Consensus, universal construction

Message-passing

- Consider a network where every two processes are connected via a reliable channel
  - ✓no losses, no creation, no duplication

- Which shared-memory results translate into message-passing?

message-passing model 是一种计算模型，其中多个独立的进程通过发送和接收消息来通信，而不是直接通过共享内存访问数据。这种模型特别适用于分布式系统和并行计算，因为它允许在没有共享内存的情况下在不同计算节点之间进行通信。

Read-write register

- Stores *values* (in a *value set* V)
- Exports two operations: read and write
  - ✓Write takes an argument in V and returns ok
  - ✓Read takes no arguments and returns a value in V

Space of registers

- Values: from binary (V={0,1}) to multi-valued
- Number of readers and writers: from 1-writer 1-reader (1W1R) to multi-writer multi-reader (NWNR)
- Safety criteria: from safe to atomic

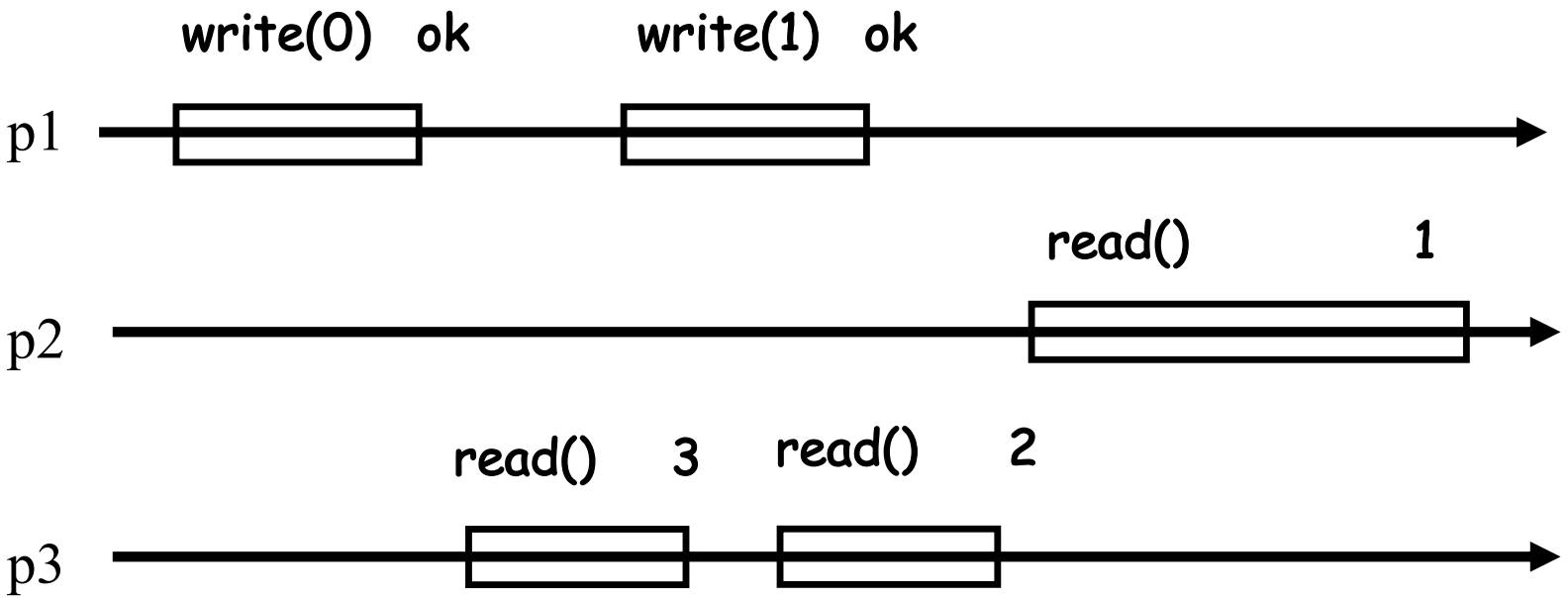
Safety criteria

- Safe registers: every read that does not overlap with a write returns the last written value
- Regular registers: every read returns the last written value, or the concurrently written value

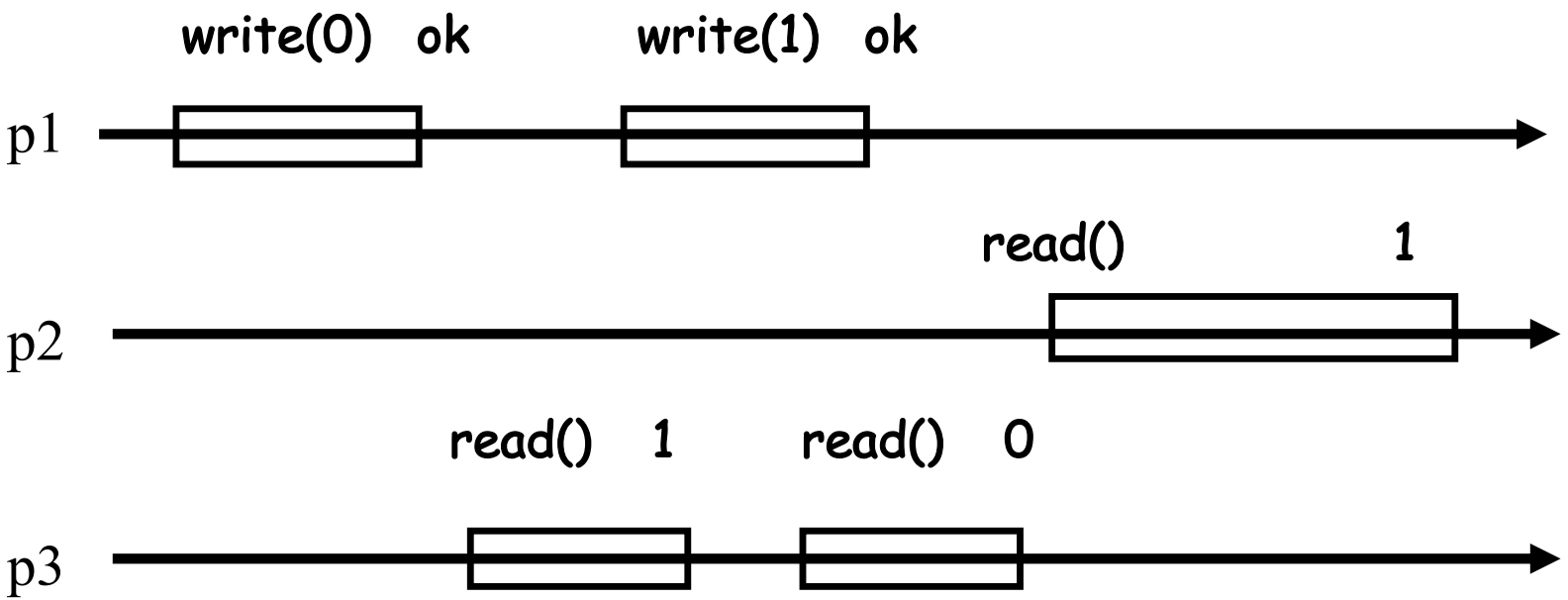
(assuming one writer)

- Atomic registers: the operations can be totally ordered, preserving legality and precedence (linearizability)
  - ✓≈ if read1 returns v, read2 returns v', and read1 precedes read2, then write(v') cannot precede write(v)

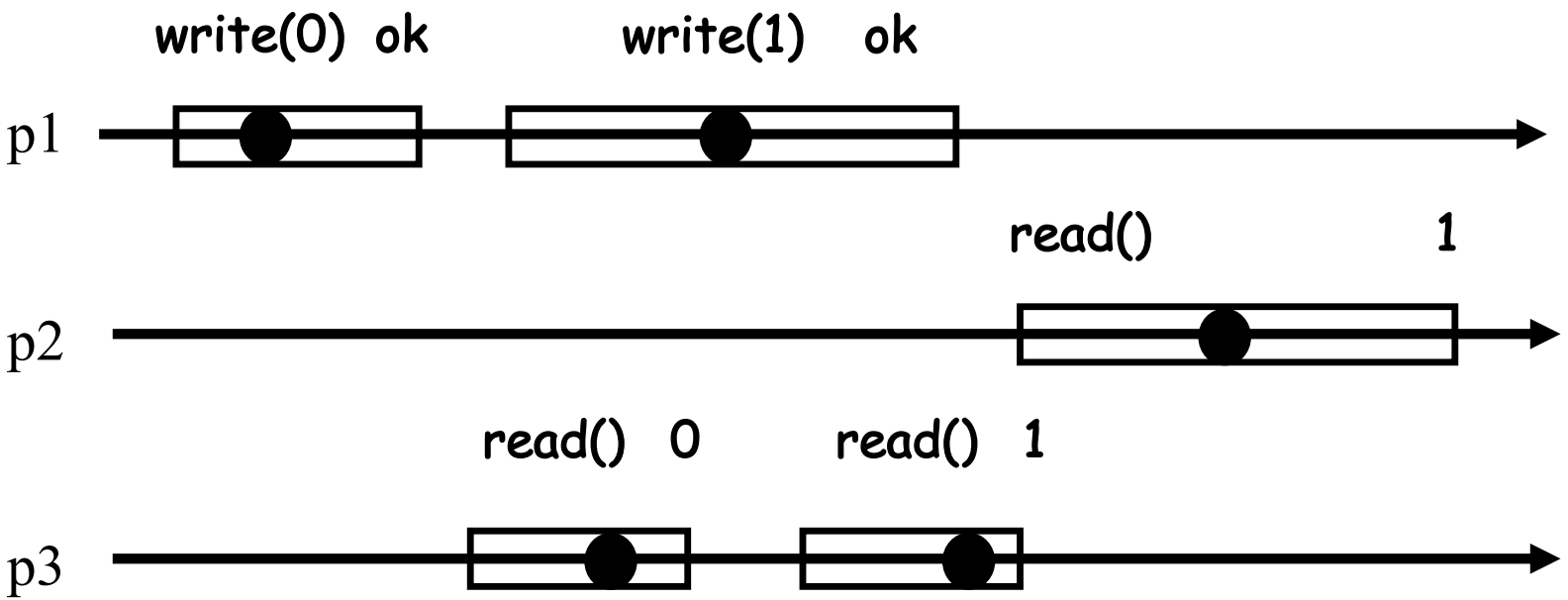
Safe register



Regular register



Atomic register



Space of registers

- Values: from binary (V={0,1}) to multi-valued
- Number of readers and writers: from 1-writer 1-reader (1W1R) to multi-writer multi-reader (NWNR)
- Safety criteria: from safe to atomic

1W1R binary safe registers can be used to implement

an NWNR multi-valued atomic registers!

Implementing message-passing in shared memory

Theorem 1 A reliable message-passing channel between two processes can be implemented using two one-writer one-reader (1W1R) read-write registers

Corollary 1 Consensus is impossible to solve in an asynchronous message-passing system if at least one process may crash

FLP: no deterministic consensus protocol can guarantee progress in an asynchronous network where at least one node may fail-stop (crash)

The ABD algorithm: implementing shared memory

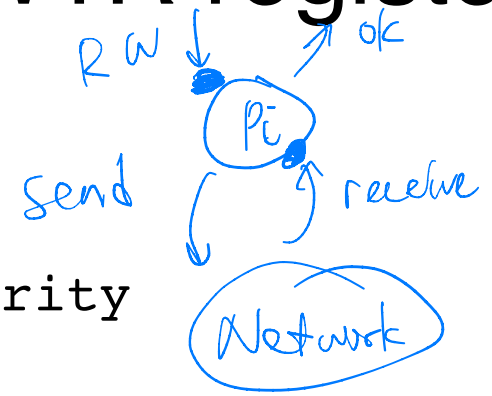
f : max # of faulty processes. $f < \frac{n}{2}$.

Theorem 2 A 1W1R atomic register can be implemented in a (reliable) message-passing model where a majority of processes are correct

- Every process is a replica of the implemented register

[Attiya, Bar-Noy, Dolev, JACM 1990]

Implementing an atomic 1W1R register




```
Upon write(v)
  t++
  send [v,t] to all
  wait until received [ack,t] from a majority
  return ok

Upon read()
  r++
  send [?,r] to all
  wait until received msgs of type (t',v',r) from a majority
  let v" be the received value with the highest timestamp t"
  if t">ti then
        vi := v"
        ti := t"
  return vi
```

ti, vi = local estimated of the written value

the most "recent" value

Implementing a 1W1R register, contd.

```
Upon receive [v,t]
  if t>t_i then
        v_i := v
        t_i := t
        send [ack,t] to the writer

Upon receive [?,r]
  send [t_i,v_i,r] to the reader
```

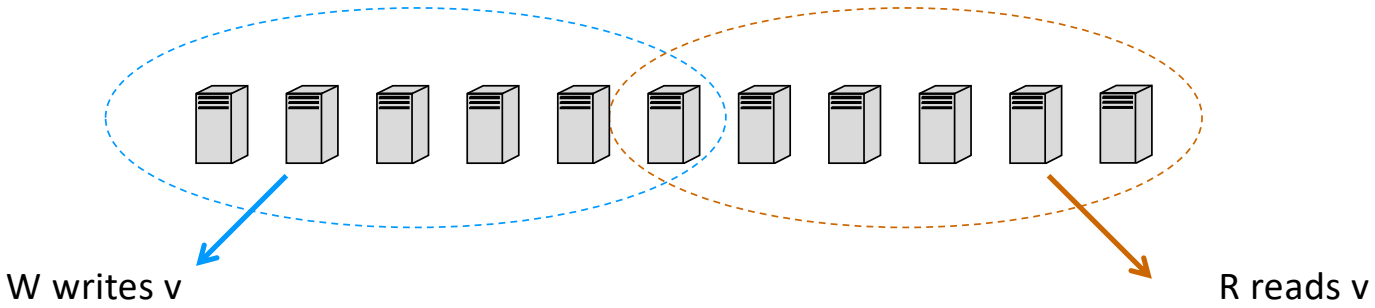
Quiz 1

- Show that the ABD algorithm executed by one writer and multiple readers implements a regular but not atomic register
- Turn the algorithm into an atomic 1WNR one
- An atomic NWNR?

Is a correct majority necessary?

Without it the reader might miss the latest written value

The quorum (set of involved processes) of any write operation must intersect with the quorum of any read operation:



Quorum systems

Let P be the set of processes

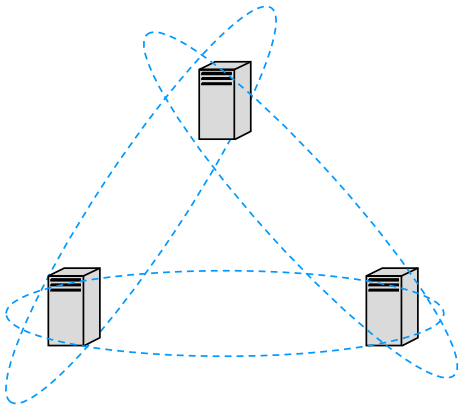
A quorum system on P is a tuple $(W_P, R_P), W_P \subseteq 2^P, R_P \subseteq 2^P$

Safety:

- $\forall W \in W_P, \forall R \in R_P: W \cap R \neq \emptyset$

Liveness:

- Some $W \in W_P$, $R \in R_P$ contains only correct processes



Example: t-resilient n-process, t<n/2

$W_P = R_P = \{S \in 2^P : |S| = n - t\}$

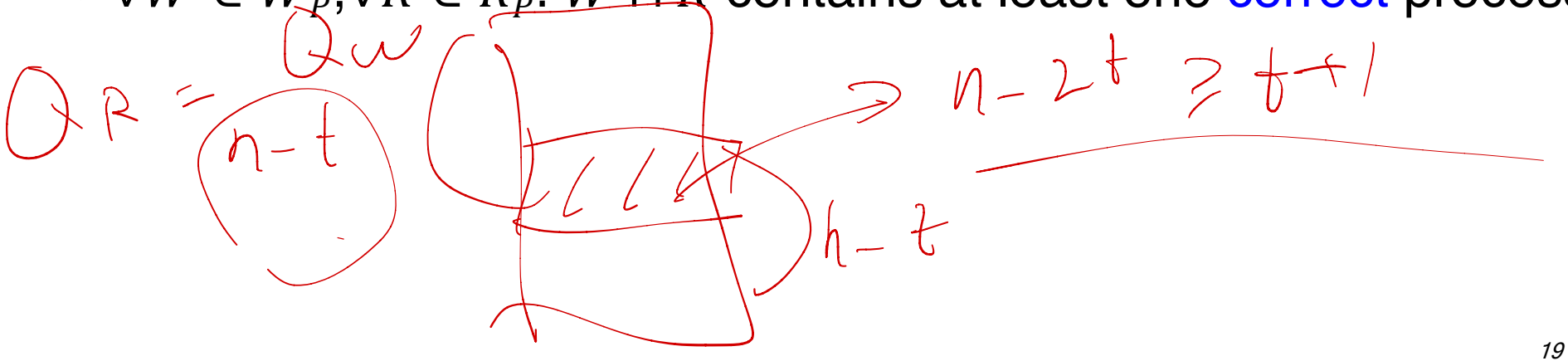
Implementing a 1W1R register with quorums

```
Upon write(v)
  t++
  send [v,t] to all
  wait until received [ack,t] from a write quorum
  return ok

Upon read()
  r++
  send [?,r] to all
  wait until received msgs of type (t',v',r) from a read quorum
  let v" be the received value with the highest timestamp t"
  if t">t_i then
        v_i := v"
        t_i := t"
  return v_i
```

Quiz 2

- For a fault-free system, design a read-optimized quorum system:
  - ✓A read operation involves a single replica
- In a t-resilient system, design a quorum system ensuring a stronger property (in every run):
  - ✓$\forall W \in W_P, \forall R \in R_P$: $W \cap R$ contains at least one correct process

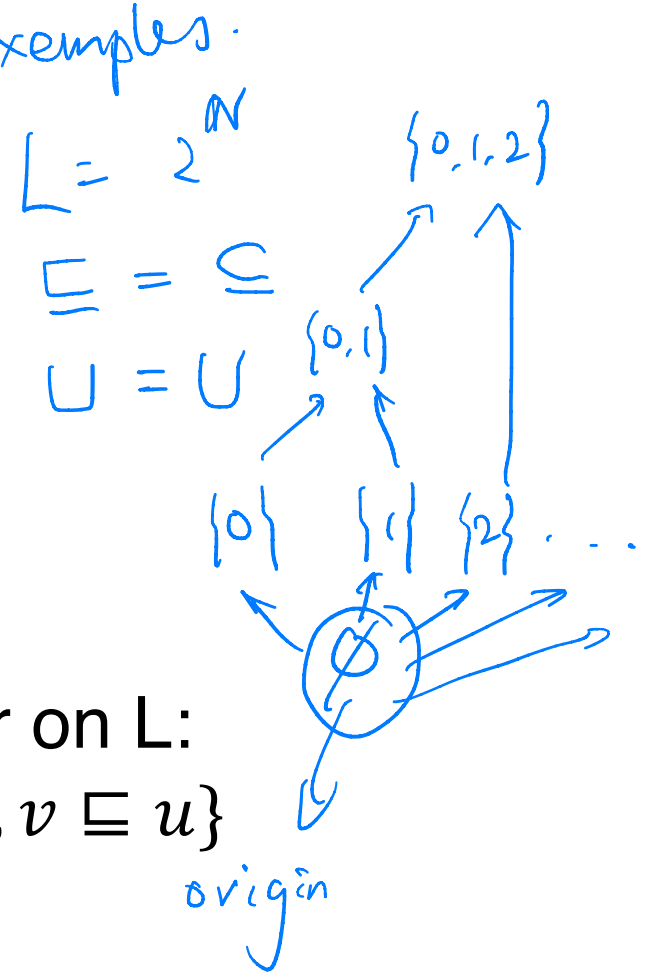


Generalizing reads-and writes

(join semi) lattices

$$(L, \sqsubseteq, \sqcup)$$

- L is a set of values
- $\sqsubseteq$ partial order on L ($\sqsubset$ *is* $\sqsubseteq \wedge \neq$)
- $\sqcup$ join (least upper-bound) operator on L:
  $\forall V \subseteq L, \sqcup V = \min\{u \in L : \forall v \in V, v \sqsubseteq u\}$
- Origin u_0: $\forall u \in L : u_0 \sqsubseteq u$

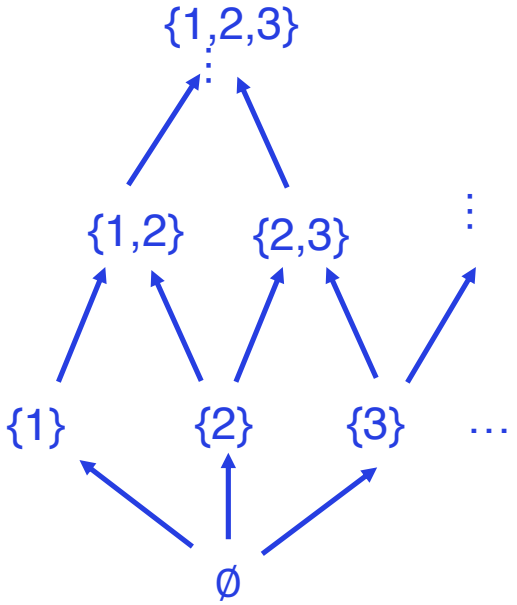


Set

A set of values with operations *add* and *read*

$$(Lset, \sqsubseteq_{set}, \sqcup_{set})$$

- $L_{set} = 2^{\mathbb{N}}$
- $\sqsubseteq_{set} = \subseteq$
- $\sqcup_{set} = \cup$
- $\emptyset$



A remove operation can be represented as $add(-v)$, assuming $L_{set} = 2^{Z}$

$\{0,1,2,-1\} \Rightarrow \{0,2\}$

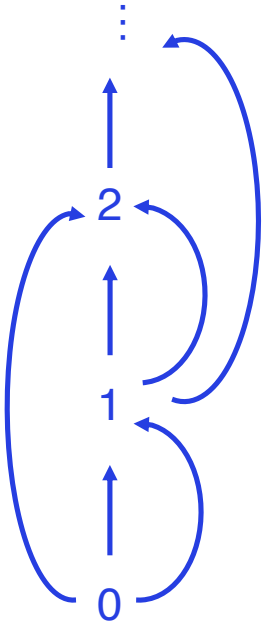
every node has a Max Register !

Max register

Read-write variable: every *read* returns the largest written value

$$(LMR, \sqsubseteq_{MR}, \sqcup_{MR})$$

- $L_{MR} = \mathbb{N}$
- $\sqsubseteq_{MR} = \leq$
- $x \sqcup_{MR} y = \max(x, y)$
- 0

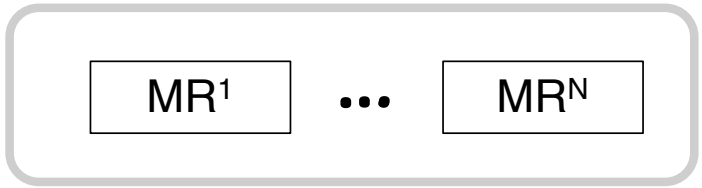


Composed lattice

$$(L, \sqsubseteq, \sqcup) = (L_1, \sqsubseteq_1, \sqcup_1) \times (L_2, \sqsubseteq_2, \sqcup_2)$$

- $L = L_1 \times L_2$
- $(x_1, x_2) \sqsubseteq (y_1, y_2) \Leftrightarrow x_1 \sqsubseteq_1 y_1 \wedge x_2 \sqsubseteq_2 y_2$
- $(x_1, x_2) \sqcup (y_1, y_2) = (x_1 \sqcup_1 y_1, x_2 \sqcup_2 y_2)$

E.g.,N max registers: $\times_{i=1,\ldots,N} (L^i_{MR}, \sqsubseteq^i_{MR}, \sqcup^i_{MR})$

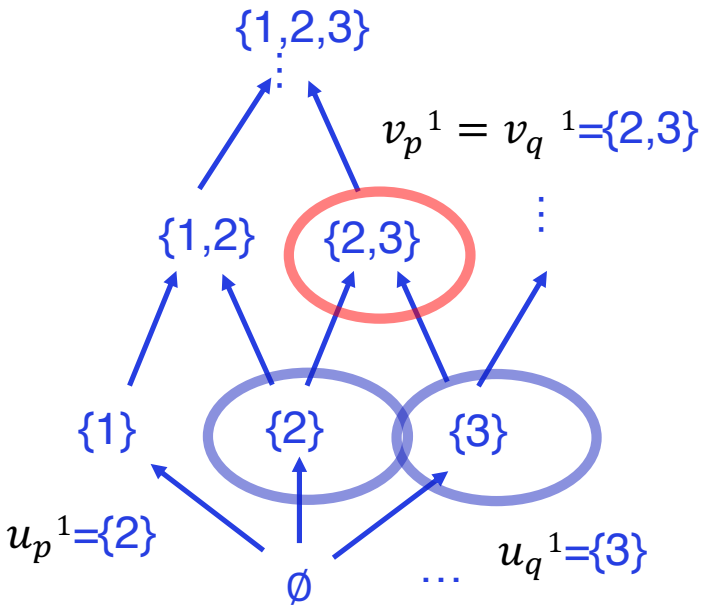


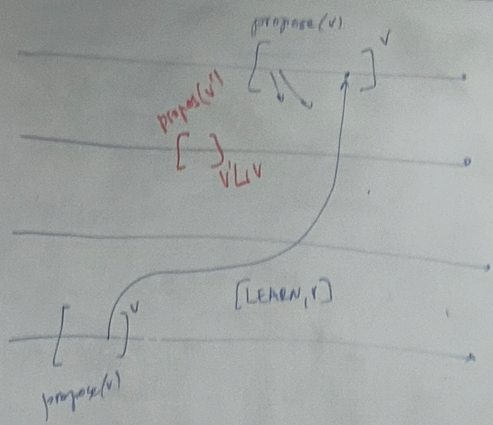
Lattice agreement on $(L, \sqsubseteq, \sqcup)$

One operation *propose* that takes a value in L (a *proposed* value) and returns a value in L (a *learned* value)

- $u_p{}^1, u_p{}^2, \ldots$ - the sequence of values proposed by p
- $v_p{}^1, v_p{}^2, \ldots$ - the sequence of values learned by p

Only consider well-formed runs: a process never invokes a *propose* operation before the previous one returns



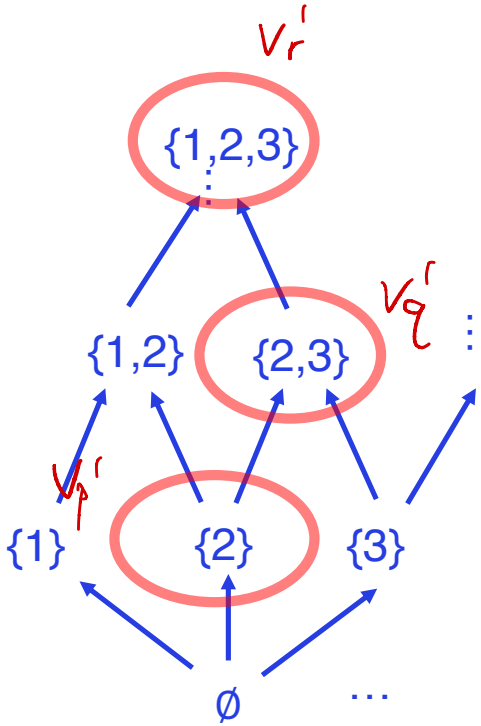

propose(v)
v
propose(v')
v' ≤ v
[LEARN, v]
propose(v)

Lattice agreement on $(L, \sqsubseteq, \sqcup)$

- Comparability: every two learnt values are *comparable*

$$\forall p, q, i, j: v_p{}^i \sqsubseteq v_q{}^j \vee v_q{}^j \sqsubseteq v_p{}^i$$

(Learnt values form a totally ordered set)



Lattice agreement on $(L, \sqsubseteq, \sqcup)$

- Comparability:

$$\forall p, q, i, j: v_p{}^i \sqsubseteq v_q{}^j \vee v_q{}^j \sqsubseteq v_p{}^i$$

- Validity: every learnt value is the join of a subset of proposed values including already committed ones

Let $V_p{}^i$ denote the values that were learned before u_p^i was proposed ($\sqcup V_p^i$ - their join)

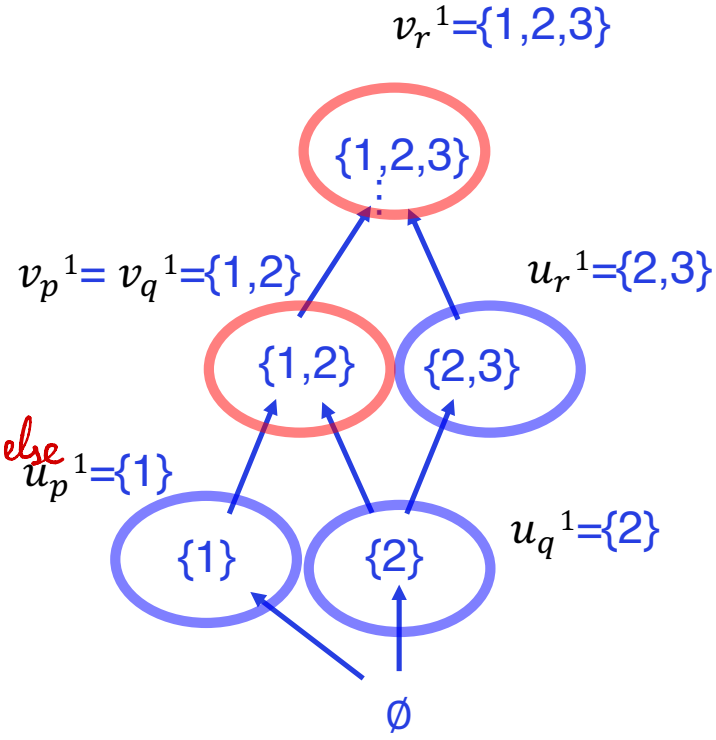
$$\forall p, i: (v_p{}^i \sqsubseteq \sqcup_{q,j} u_q{}^j) \wedge (u_p{}^i \sqsubseteq v_p{}^i) \wedge (\sqcup V_P^i \sqsubseteq v_p{}^i)$$

proposed i by p should return a proposed value ∪ sth else

all proposed values

joint of the learnt set

(No "bogus" value can be learnt)



Lattice agreement on $(L, \sqsubseteq, \sqcup)$

- Comparability: $\forall p, q, i, j: {v_p}^i \sqsubseteq {v_q}^j \vee {v_q}^j \sqsubseteq {v_p}^i$
- Validity:$\forall p, i: ({v_p}^i \sqsubseteq \sqcup_{q,j} {u_q}^j) \wedge ({u_p}^i \sqsubseteq {v_p}^i) \wedge (\sqcup V_P^i \sqsubseteq {v_p}^i \quad)$
- Liveness: every propose operation invoked by a correct process eventually returns

Implementing LA

To propose v and learn a new value:

Try to ensure that a quorum accepts v or there is a learnt value v' such that $v \sqsubseteq v'$

A replica accepts a value only if it accepted no incompatible values before

After at most N+1 attempts, either the current proposal is installed in a quorum (and learnt) or a concurrent value containing v is installed

Quorum intersection => safety

Acceptors acknowledge proposed values => Liveness

≈ helping like in atomic snapshots

One-shot case:

currentValue = u_0.

- propose(w):

  proposedValue = u

  while (True):

    send [PROPOSE, proposedValue, p] to all.

    wait until receive (ACK, v_q) from aquorum.

    if $\bigcup_q v_q$ = proposedValue,

      currentValue = proposedValue $\cup$ currentValue

      return proposedValue

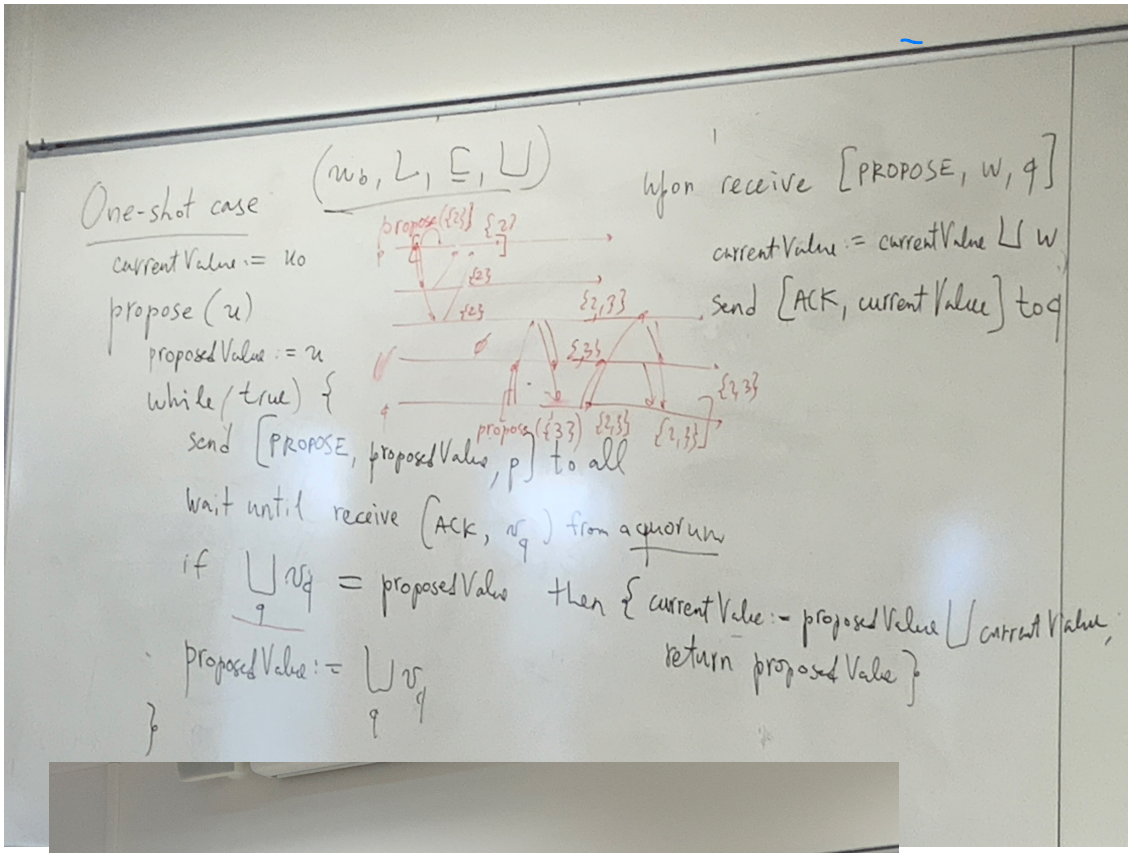
    proposedValue = $\bigcup_q v_q$

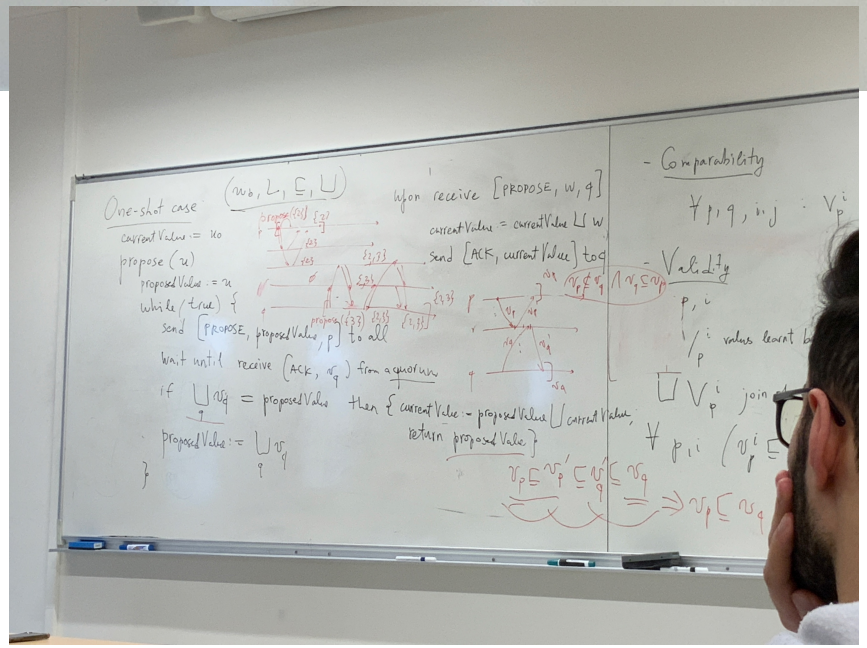
- upon receive [PROPOSE, w, q]:

  currentValue = currentValue $\cup$ w.

  send [ACK, currentValue] to q

Algorithm Basics of Blockchains ch1 - 4





Implementing LA

```
Local variables:
  bufferedValue = u_0 // join of known proposals
  proposedValue = u_0// the current proposal
  acceptedValue = u_0 // join of accepted proposals
  readyToLearn = false // ready to adopt a learn value

Upon propose(v) // process p proposes v
  t++ // sequence number of the proposal
  readyToLearn := false
  proposedValue := v
  bufferedValue := bufferedValue ⊔ v
  send [PROPOSE,v,t,p] to all
```

To read the "current value": propose(u_0) – the lattice origin

Implementing LA (2)

```
Upon received [PROPOSE,v',t',p']
  if acceptedValue ⊑ v' then
        acceptedValue := v'
        send [ACK,v',t',p'] to p'
        // accept the proposal
  else
        acceptedValue := acceptedValue ⊔ v'
        send [NACK,acceptedValue,t',p'] to p'
        // reject the proposal
```

Implementing LA (3)

```
Upon received [ACK/NACK,v',t,p] from a quorum
  if no [NACK,v',t,p] received then
        send [LEARN,bufferedValue] to all
        return bufferedValue// learn a new value
  else
        t++
        if not readyToLearn then
           readyToLearn := true
           proposedValue := bufferedValue
        send [PROPOSE,bufferedValue,t,p] to all
        // send a new proposal
```

Implementing LA (4)

```
Upon received [NACK,v',t,p]
  bufferedValue := bufferedValue ⊔ v'


Upon received [LEARN,v']
    if proposedValue ⊏ v' and readyToLearn then
        send [LEARN,v'] to all // "reliable" broadcast
        return v' // adopt a new value
```

- consensus.
- consensus number (how to prove).
- LA.
- storage and message passing (RW memory can be implemented in message-passage in quorums) ABD

ata Type

CRDT

- (Deterministic) fault-tolerant consensus is impossible in asynchrony (FLP, valencies)
- Consensus is universal
- Lattice Agreement is useful
- RW memory can be implemented in message-passing (with quorums) ABD

Commit-Adopt

propose (v), $v \in V$

(v', f), $v' \in V$, $f \in \{true, false\}$ (committed, adopted)

- Liveness: wait-free with probability 1
- Validity: v' was proposed
- Agreement → No two processes decide with different values
  - if no two processes propose different values, then no process returns $(v', false)$
  - if a process returns $(v', true)$ no process returns $(v'', *)$, such that $v'' \neq v'$

Las vegas

Monte-carlo

Application 1: timestamped value

Max-register on $\mathbb{N}\times V$ (V is an ordered set)

$$(LTV, \sqsubseteq_{TV}, \sqcup_{TV})$$

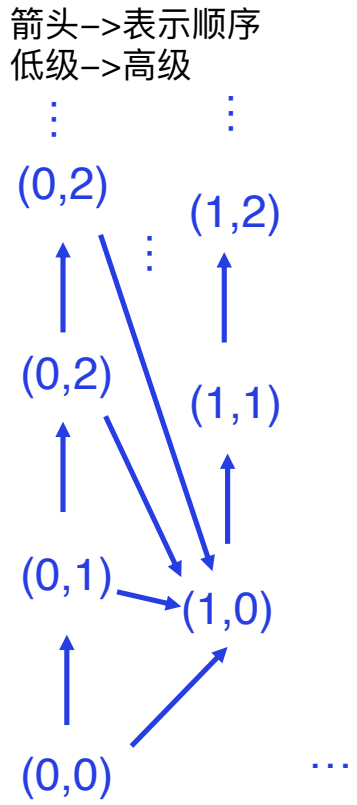
- $L_{TV} = \mathbb{N}\times V$ *(timestamp, value)*
- $\sqsubseteq_{TV} = (\leq_{\mathbb{N}}, \leq_{V})$ lexicographic total order
- $(t, v) \sqcup_{TV} (t', v') =$ the tuple with the highest timestamp $\max(t, t')$ (highest value if $t = t'$)
- $u_0 = (0,0)$

To read the value with the highest timestamp:

propose$(u_0) \rightarrow (t, v)$

To write a value v with timestamp t:

propose$((t, v))$



Application 2: atomic snapshot

(m-position multi-writer)

- Each process p_i is provided with operations:
  - ✓update(v,j), returns ok
  - ✓snapshot(), returns $[v_1,\ldots,v_m]$
- In a sequential execution:

  For each $[v_1,\ldots,v_m]$ returned by $snapshot_i()$, v_j (j=1,…,m) is the argument of the last update(.,j)

  (or the initial value if no such update)

In a concurrent system, the lattice agreement is used to determine the order of updates seen by the snapshots, providing the following guarantees:

Comparability: Any two snapshots are comparable. That is, one snapshot is either a predecessor or successor of another snapshot or they are the same, reflecting the partial order of updates.

Implementing atomic snapshot with LA

Lattice agreement on the composition of m timestamped values

$$(\mathrm{L}, \sqsubseteq, \sqcup) = \times_{i=1,\ldots,m} (L_{TV}^{i}, \sqsubseteq_{TV}^{i}, \sqcup_{TV}^{i})$$

$snapshot()$:

- $propose(u_0) \rightarrow [(t_1, v_1), \ldots, (t_m, v_m)]$

$update(v, j)$:

取得最新的状态

- $propose(u_0) \rightarrow [(t_1, v_1), \ldots, (t_j, v_j), \ldots, (t_m, v_m)]$
- $propose([(t_1, v_1), \ldots, (t_j + 1, v), \ldots, (t_m, v_m)])$

propose new timestamp and value

Snapshots are totally ordered!

Contain all earlier updates

Asset transfer ("cryptocurrency")

State:

- P - set of processes
- A - set of *accounts*
- μ: A$\rightarrow 2^{P}$ - *ownership map* (we assume single owner: A$\rightarrow$P)
- β: A$\rightarrow$**N** – balance map (β_0 – initial balances)

Interface:

- *transfer*(a,b,x) – called by an owner of a, returns a boolean (success or failure)
- *read*(a) – returns the balance

We call it asset transfer data type

Application 3: asset transfer

LA on the composition of n sets

$$\times_{i=1,\ldots,n} (L^i_{set}, \sqsubseteq^i_{set}, \sqcup^i_{set})$$

(sets of outgoing transfers for each account)

$transfer(a, b, x)$:

- Read the state, check the balance of a
- Add (b, x) to set a
- Propose the new state

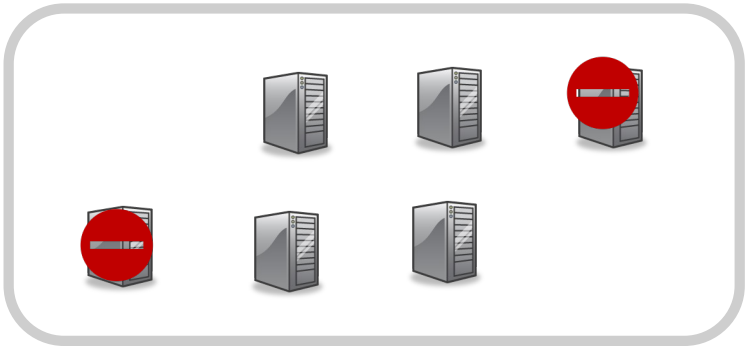
Total order of sets implies atomicity of transfers

在asset transfer应用中，LA用于确保资产转移的原子性。每个账户的转账操作可以通过LA来表示，以保证所有转账操作的顺序一致，避免了如双重支付等问题。每次转账操作都被表示为一个在特定账户集上的lattice操作，并通过LA协议来确保所有操作的全局顺序一致性。这样，即使在异步环境中，转账的原子性也得到了保证。

在update操作中，可能需要进行两次propose，第一次是为了读取当前状态，确保提案值与当前值兼容；第二次则是为了提交新的状态。这个过程确保了所有值的顺序一致性，并且保证了任何时刻对snapshot的调用都能够返回一个一致的系统状态视图。

Using LA: reconfiguration

A set of replicas of a lattice agreement protocol may be reconfigured over time



Reconfiguration?

A set of replicas of a lattice agreement protocol may be reconfigured over time

Reconfiguration in storage systems:

- consensus-based [Rambo'02,03,10]
- asynchronous [Dynastore'11,…]

Configuration lattice

Abstract lattice $(Cf, \sqsubseteq_{cf}, \sqcup_{cf})$

Every element $C \in C_f$ carries:

- $C.members$ – a set of replicas
- $C.quorums \subseteq 2^{C.members}$
- Other attributes (used in $\sqcup_{cf}$)

Configuration lattice: example

$$(Cf, \sqsubseteq_{cf}, \sqcup_{cf})$$

- Elements of Cf are of type {+1,+2,+3,-2}
- $\sqsubseteq_{cf} = \subseteq$ and $\sqcup_{cf} = \cup$
- $C.members$ – all added and not yet removed
- $C.quorums$ – all majorities of $C.members$

Reconfigurable object

- Solve lattice agreement on

$$(L, \sqsubseteq, \sqcup) \times (Cf, \sqsubseteq_{cf}, \sqcup_{cf})$$

- Every configuration update results in a join of proposed configurations
- Every update should be installed in a quorum of each candidate configuration
- A new decided state makes all preceding configurations obsolete

A configuration must be available as long as it is not obsolete!

Plug-and-play reconfiguration

Plug the corresponding lattice and get:

- Max register
- Set
- Atomic snapshot
- Conflict detector
- Commit-adopt
- Safe agreement
- Asset transfer
- …

Literature

- C. Cachin, R. Guerraoui, L. Rodrigues. Introduction to Reliable and Secure Distributed Programming. Springer, 2011
- N. Lynch. Distributed Algorithms. Morgan Kaufmann Publishers. 1996
- H. Attiya, A. Bar-Noy, D. Dolev: Sharing Memory Robustly in Message-Passing Systems. J. ACM 42(1): 124-142 (1995)
- H. Attiya, M. Herlihy, O. Rachman: Atomic Snapshots Using Lattice Agreement. Distributed Computing 8(3): 121-132 (1995)
- J. M. Faleiro, S. K. Rajamani, K. Rajan, G. Ramalingam, K. Vaswani: Generalized lattice agreement. PODC 2012: 125-134
- Kuznetsov et al. Reconfigurable Lattice Agreement. OPODIS 2019