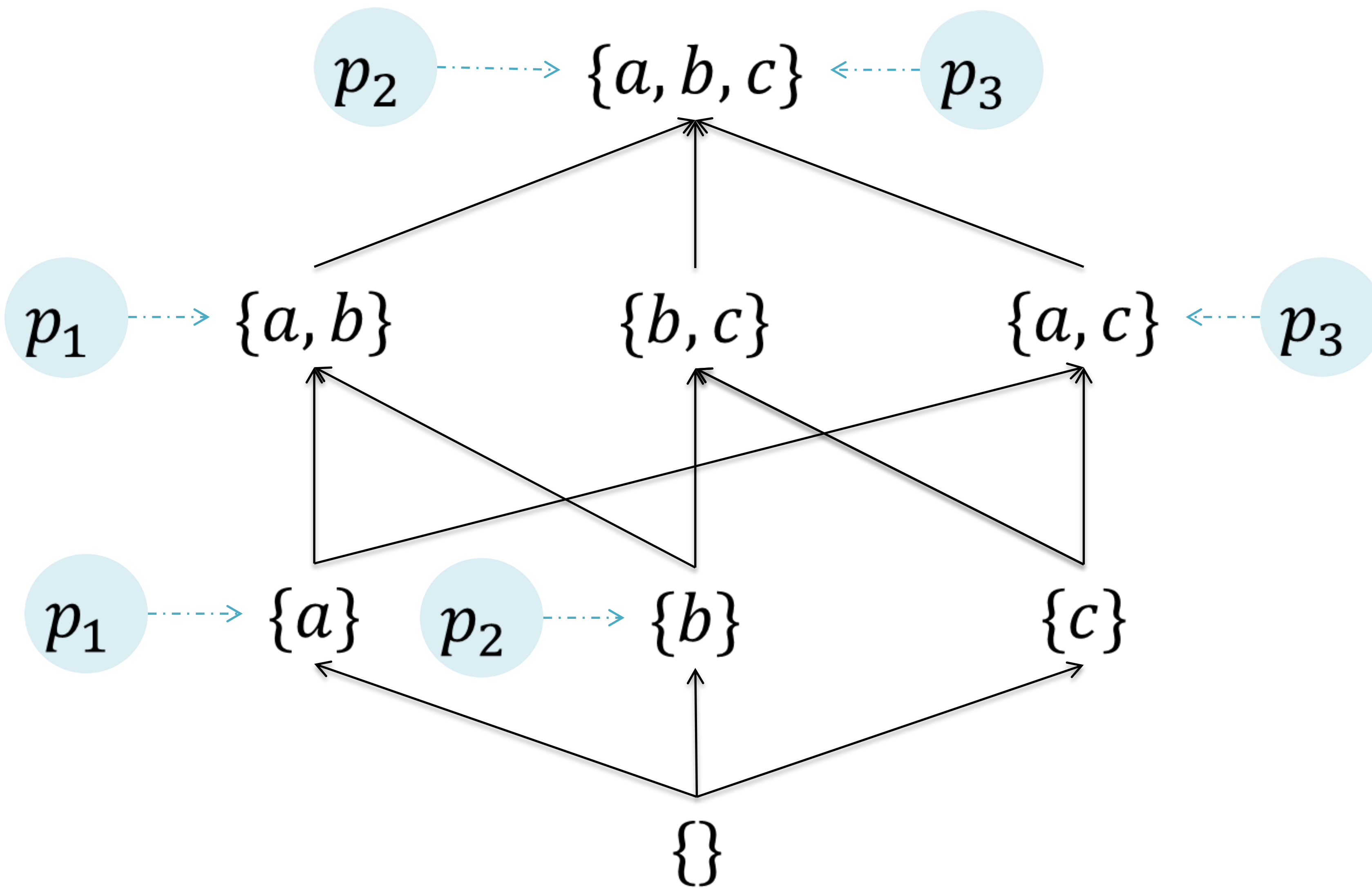

Data Structures on GLA
Start
Classification
Consensus Reduction
Partial Nullifying order
Genearlized Lattice Aggrement
Sagar Chordia
{Kapil, Rama, Kaushik}

Lattice agreement

- N processes.
- Each process starts with an input value
  $u_i \in (L, \leq, V)$ a finite join semi lattice.
  - $\leq$ is partial order
  - For any two elements $a, b \in L$. $a \vee b$ exists.
- Every non-faulty process outputs a value v_i
  - v_i is join of input values including its own.
  - Any two output values v_i and v_j are comparable i.e. either $v_i \leq v_j$ or $v_j \leq v_i$
  - Every correct process eventually outputs a value.

Lattice agreement

Distributed Asynchronous failure-prone system



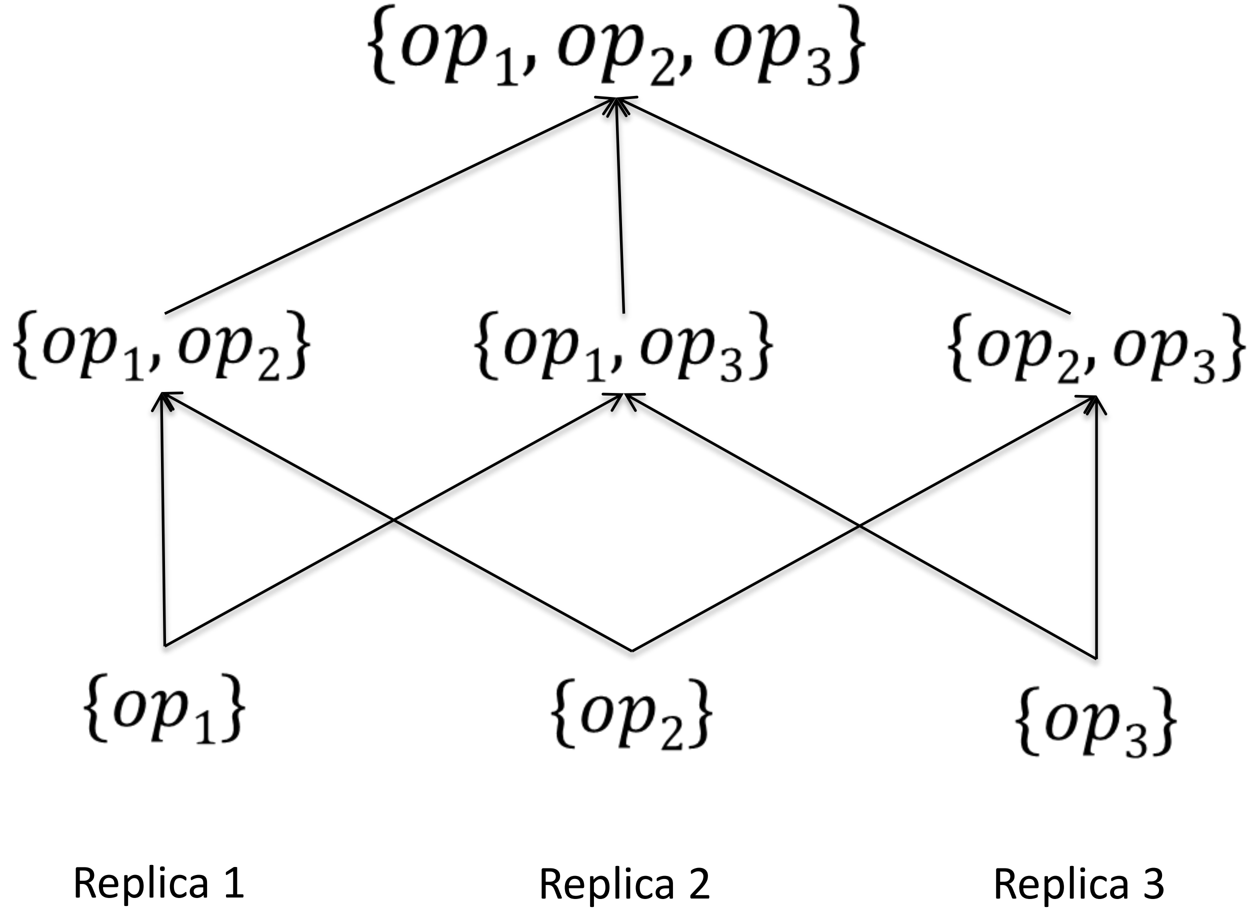
Generalized lattice agreement

- Generalization of lattice agreement
  - Process receives sequence of values u_i^j
  - Process outputs sequence of values v_i^j
  - Any two values are comparable.
  - Every value received by correct process is eventually included in an output value.
- Fault-tolerant model.
- Wait-free algorithm. $O(n)$ message delays.

Application of GLA

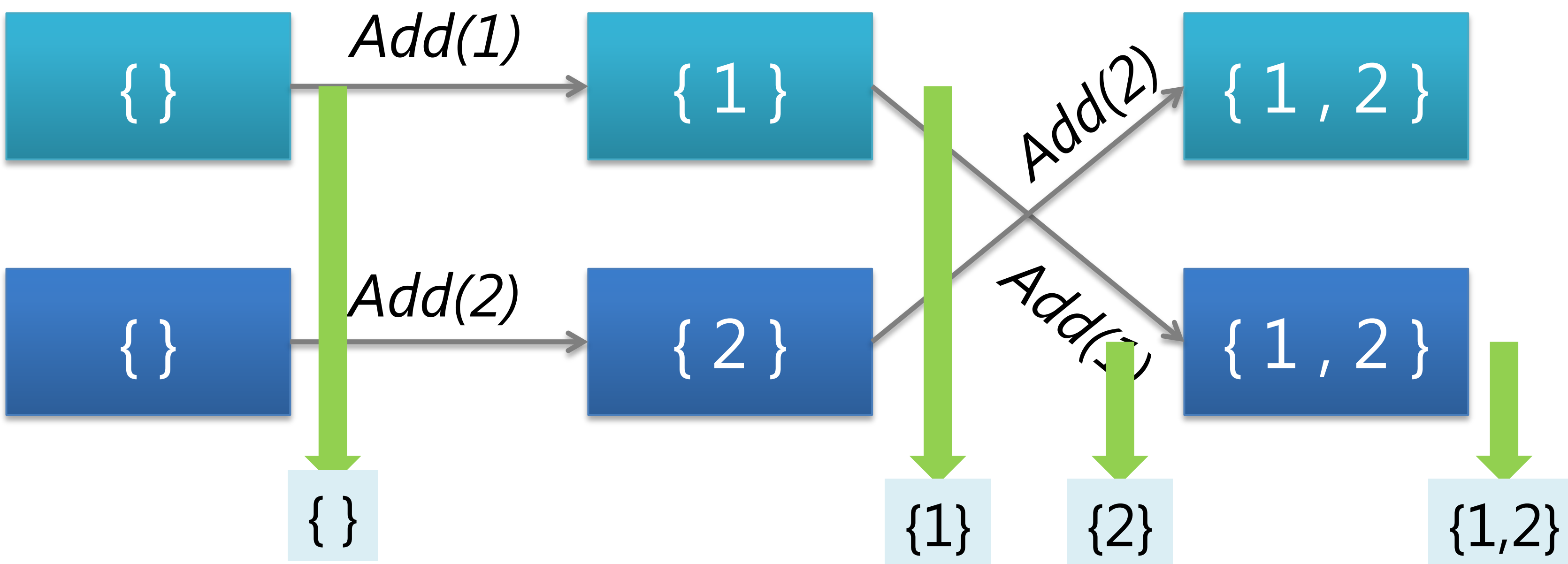
- Strong consistencies
  - Sequential consistency
    - Serializability and program order of client operations is maintained.
  - Linearizability
    - Operations appear to execute instantaneously.
- GLA can be used to implement specific-class of state machines
  - Two kind of operations (a) void update (b) read
  - All updates **commute**

SM with GLA


$\{op_1, op_2, op_3\}$
$\{op_1, op_2\}$
$\{op_1, op_3\}$
$\{op_2, op_3\}$
$\{op_1\}$
$\{op_2\}$
$\{op_3\}$
Replica 1
Replica 2
Replica 3

Commutative operations

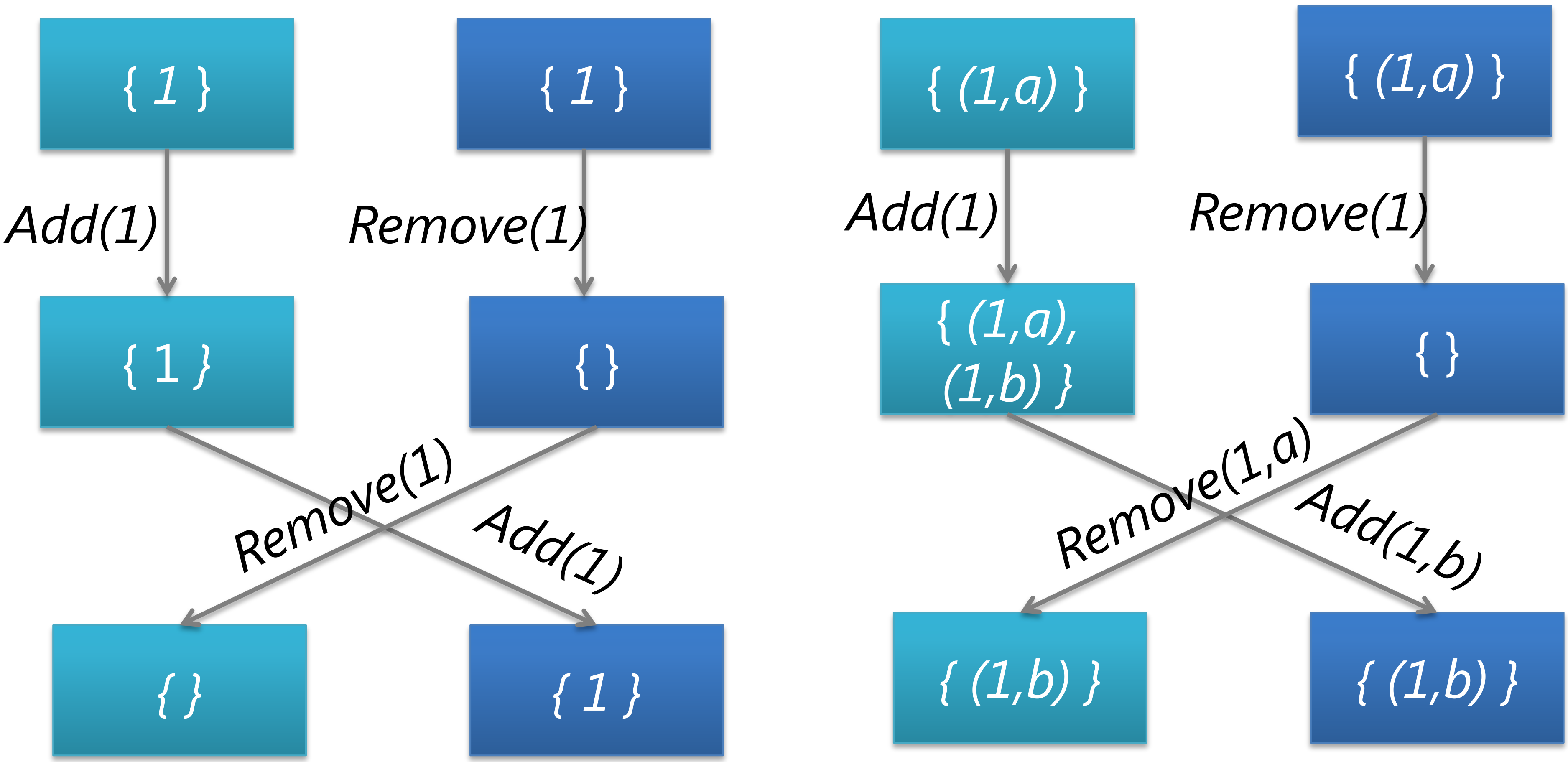
- updates commute => convergence



- Convergence ≠> Consistent Queries!
- GLA learns chain of values

{}, {2}, {1,2} ✓ {}, {1}, {1,2} ✓

Observed Remove set



Inconsistent states

Consistent states

Translator

- Mathematically, operations are functions (on some set of values).
- The translator
  - Ordering of non-overlapping operations is preserved.
  - Maps every *op* on *set S* into *op'* on a *set S'*.
  - *Op’* on set *S’* are commutative.
- Correctness
  - partially-ordered *set X* of operations on *S* should be mapped to *set Y* of operations on S‘
  - Such that the state produced by *Y* corresponds to the state produced by some linear execution of X (consistent with its partial order).

Big-picture

Client-submitted operations (O)
(can be non-commutative)

Translator

- Translated-operations (commutative)
- Statically orders non-commutative ops

PO1

Partial order 1

GLA

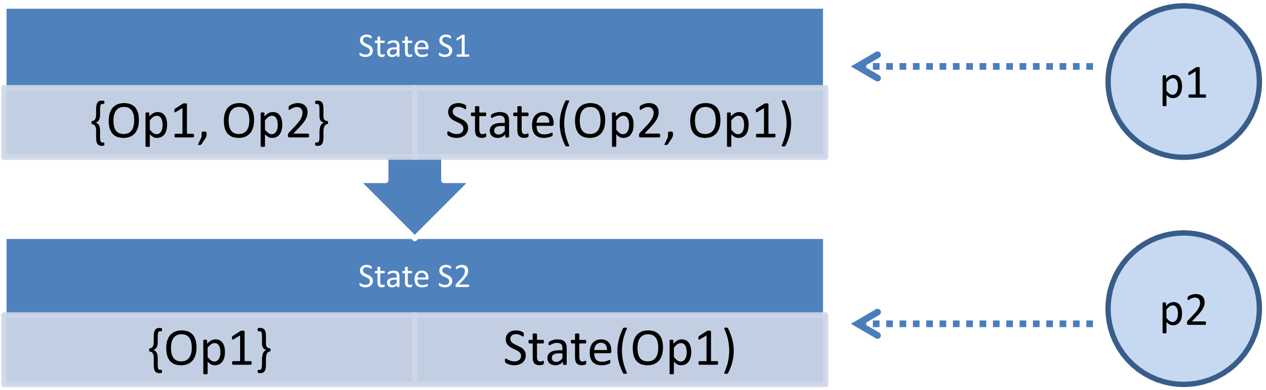
- Partial ordering of operations.
- Convergent States
- Consistent Reads

PO2

Partial order 2

What if PO1 != PO2 ?

Problem!



- Op1 and Op2 don't commute. Let Op2 < Op1 in static ordering.
- Effect:
  - State(Op2, Op1) exposes the effect of Op_2 followed by effect of Op1.
  - State(Op1) reflects effect of Op1
  - To guarantee consistency State(Op1) must reflect effect of Op2.
- Extra conditions needed?

Nullifying semantics

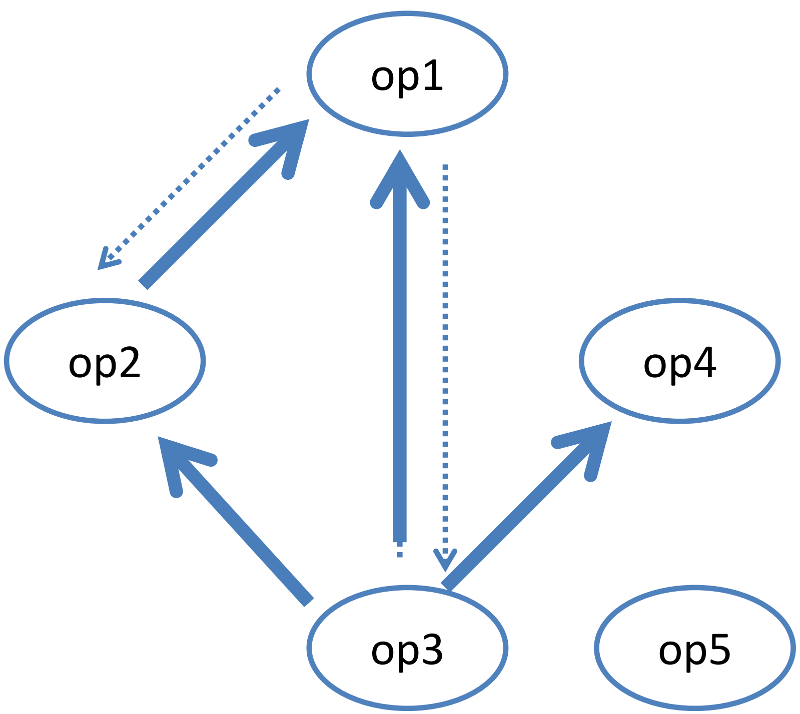
- Nullifying operations:
  - Op1(Op2(s)) = Op1(s) where s is the state.
  - Op1 is said to nullify Op2.
- In set Add(x), Remove(x) have nullifying property.
  - Add(x, Remove(x, s)) = Add(x,s)
  - Remove(x, Add(x, s)) = Remove(x,s)
- Sequence doesn’t have this.
  - {AddRight(e,x), AddRight(e,y)} != {AddRight(e,y)}

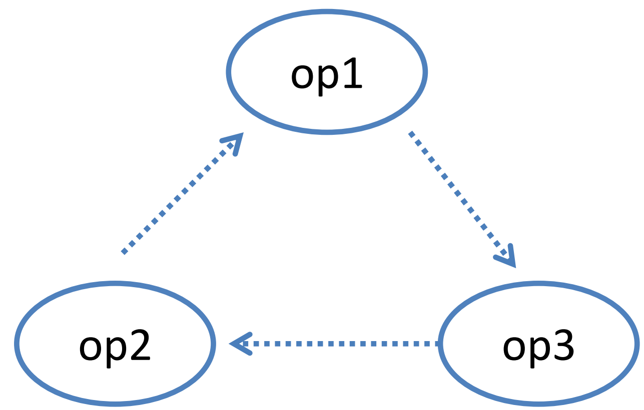
Partial nullifying order

- For every pair of non-commutative operations op_1, op_2 there can be two kind of nullifying semantics.
  - $State(op_1, op_2) = State(op_2)$
  - $State(op_2, op_1) = State(op_1)$
- But is not necessary for two operations to mutually nullify each other always.

- **Sufficient condition**: There exist a partial order < St.
  - (a) If op1 < op2, then op2 nullifies op1, and
  - (b) If op1 and op2 are incomparable in the ordering, then they commute with each other.
- Use this partial-order to give consistency

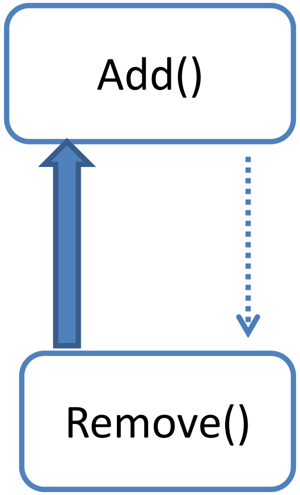
- Assume all non-commutative operations have nullifying semantics
- Consider graph, where nodes are operations
    - Edge from op_1 to op_2 if op_2 nullifies op_1
    - If no edge between op_1 and op_2 then they commute.
- **Sufficient condition** - There should exist partial ordering of nullifying operations.



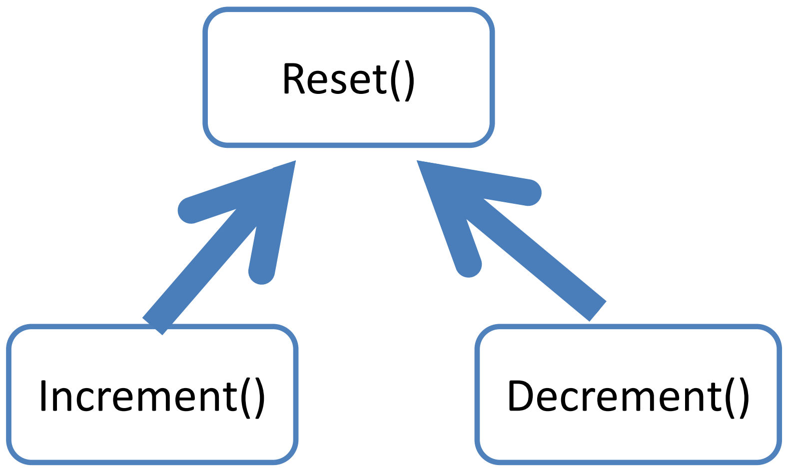


Can't use GLA

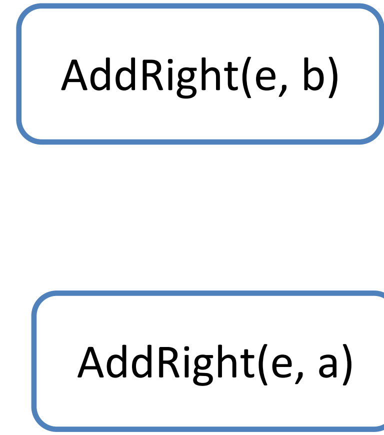
Some Data-Structures



OR-Set
(Valid)



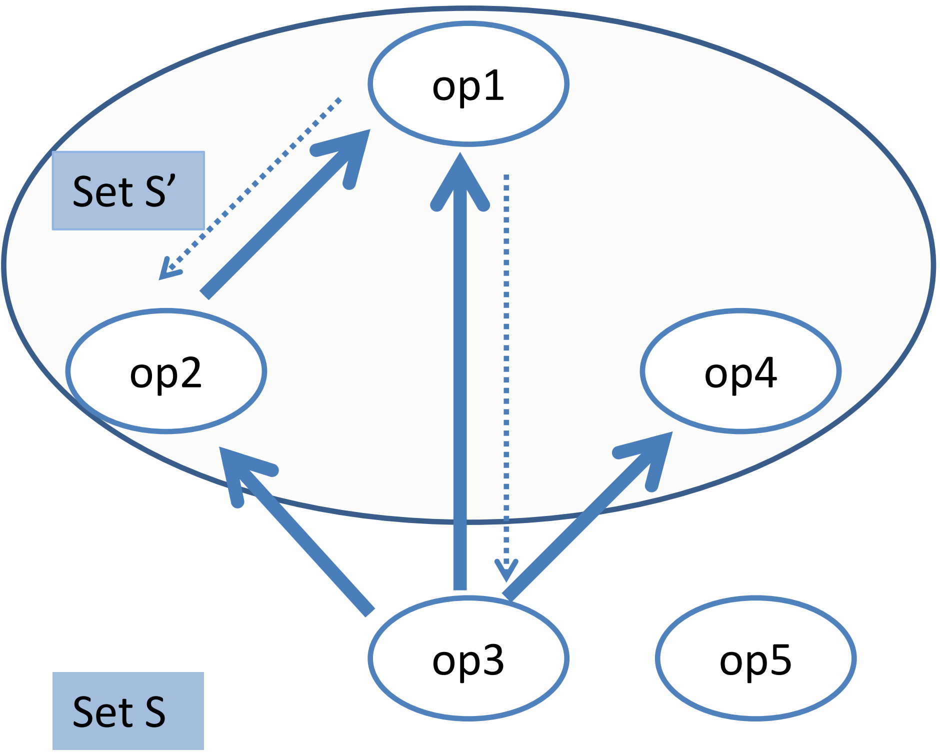
Reset-Register
(Valid)



Sequence
(Not Valid)

Why partial nullifying order works

- Let S = $\{op_1, op_2, \ldots op_n\}$. There exists partial order in S.
- Let S' = $\{op_{i1}, op_{i2} \ldots\} \subseteq S$
- To Show - value(S') is consistent with value(S)

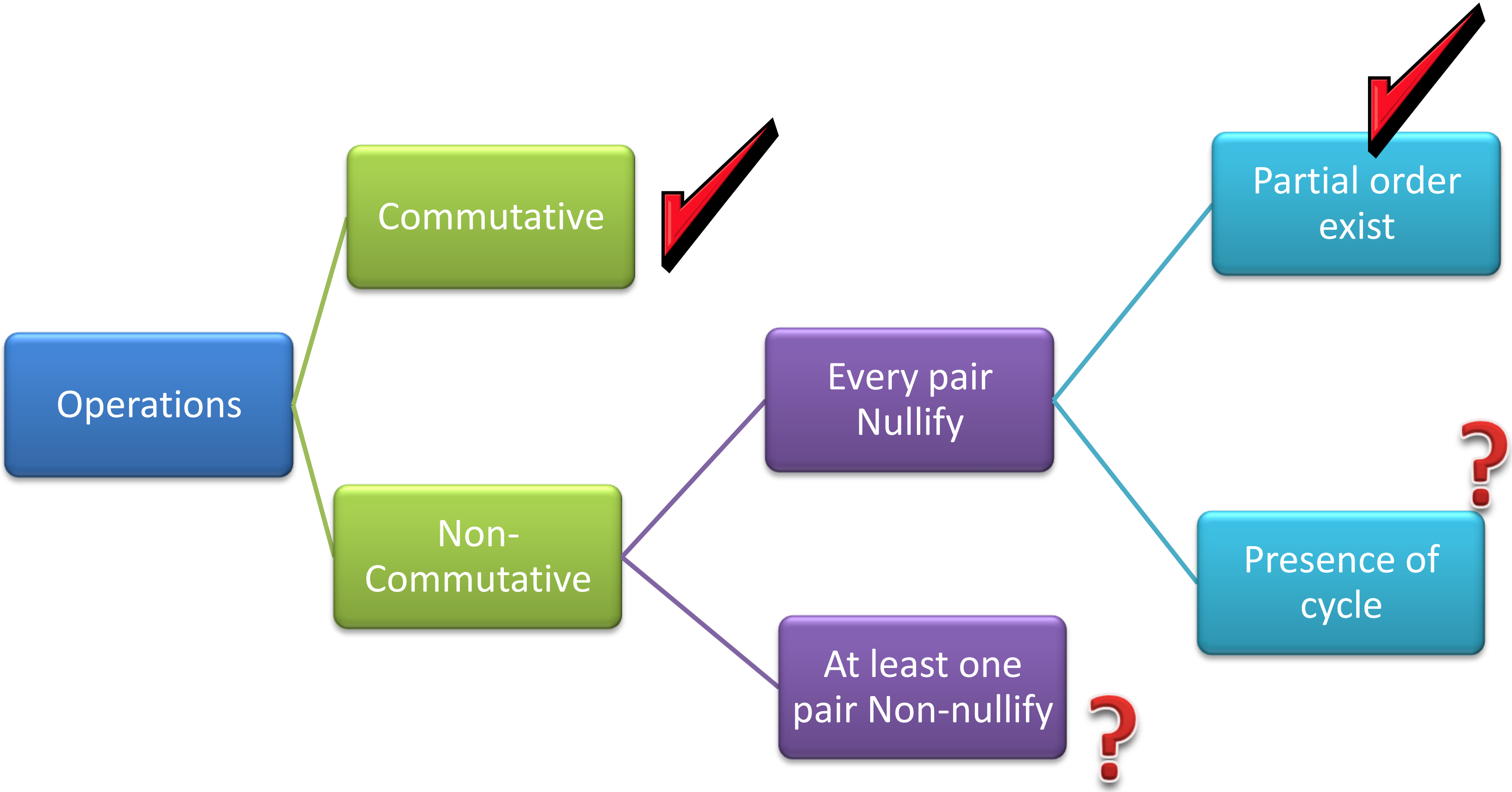


- S' = {op1, op2, op4}
- S = {op1, op2, op3, op4, op5}
- S' ⊆ S

- Define dc(S, S') := downward closure of set S with respect to S'.
  dc(S,S')={op1,op2,op3,op4}

- Value(S') = Value(dc(S, S'))
- Nullifying semantics guarantee this.

Classification



Consensus

- n processes
- Each process proposes value and they have to agree on common value, one of proposed value.

- CAP theorem: Impossible for distributed systems to have following three guarantees simultaneously
  - Consistency (all nodes see same data at same time)
  - Availability (response for every request)
  - Partition tolerance (fault-tolerant)

- two-consensus: two process have to decide on common value. Un-decidable Problem!

Possible states

- Consider Op_1, Op_2 as only operations proposed to GLA by two processes
  - Possible states:
    - Op_1
    - Op_2
    - Op_1, Op_1
    - Op_2, Op_2
    - Op_1, Op_2
    - Op_2, Op_1
- If Op_1 is proposed, Linearisability guarantee gives
  - Possible reads:
    - Op_1
    - Op_1, Op_2
    - Op_1, Op_1
    - Op_2, Op_1
- If Op_2 is proposed, Linearisability guarantee gives
  - Possible reads:
    - Op_2
    - Op_2, Op_2
    - Op_2, Op_1
    - Op_1, Op_2

Reduction of consensus to GLA

C_Propose(value v)

- If(v == 1)

```
GLA_Propose(op₁)
S  = GLA_read()

If S ∈ {state(op₁), state(op₁,op₂), state(op₁,op₁)}
        C_Learn(1);
Else                                    //{state(op2,op1)}
         C_learn(0);
```

- Else if (v == 0)

```
GLA_Propose(op₂)
S = GLA_read()

If S ∈ {state(op₂), state(op₂,op₁), state(op₂,op₂)}
        C_Learn(0);
Else                                    //{state(op₁,op₂)}
         C_learn(1);
```

Process proposing op1

State Read		Possible States
State(op_1)	**SET A**	Sate(op_1, op_2), State(op_1, op_1)
State($op_{1,}$ op_2)		Sate(op_1)
State($op_{1,}$ op_1)		Sate(op_1)
State($op_{2,}$ op_1)	**SET B**	State(op_2)

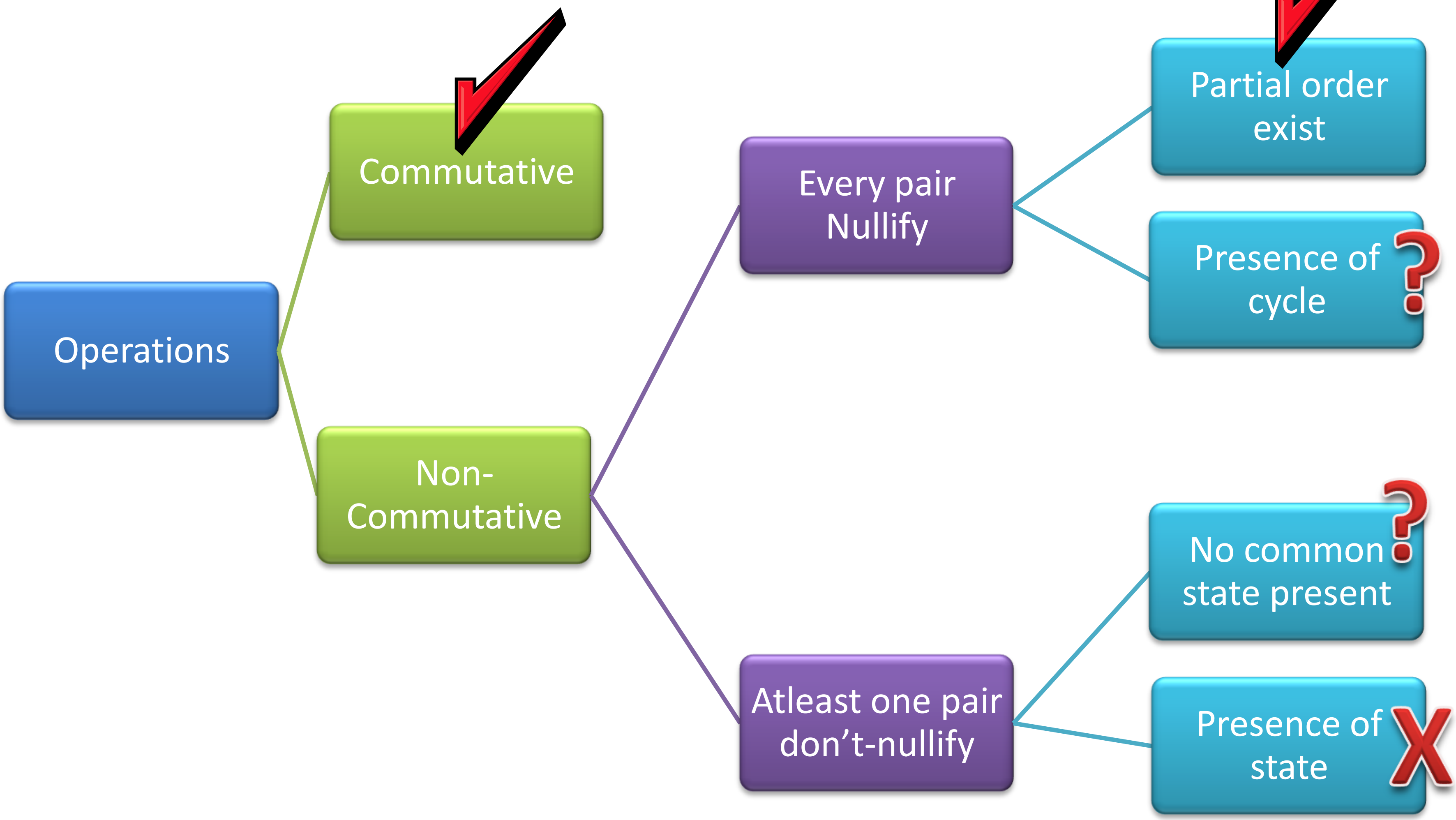
State(op2)	!=	State(op1)	#Initial conditions
	!=	State(op1, op2)	#Non-nullifying
	!=	State(op1, op1)	#Not possible
State(op2, op1)	!=	State(op1)	#Non-nullifying
	!=	State(op1, op2)	#Non-commutative e
	!=	State(op1, op1)	#Additional constraints

- Set A and B don’t have any common value provided
  - Operations are non-commutative and non-nullifying.
  - **(Extra Assumption)** There exists state where all previous inequalities hold.

- So given a state we can find out its set.
- Knowing set => unique value can be chosen.
  - SET A: value 1
  - SET B: value 0

- Thus we are able to solve 2-consensus using GLA instance.
- As 2-consensus is non-wait free => GLA can’t be non-wait free

Classification



Summary

- GLA is computational model to get strong consistency guarantees in distributed systems
- For data-structures to work with GLA:
  - Necessary condition: Every pair of non-commutative operations should have nullifying semantics.
  - There should exist partial nullifying order

Thanks!

- References:
  - Generalized lattice agreement by *Jose, Sriram, Kaushik, Rama, Kapil*
  - GLA and Data-Structures implementation *Hari, Sagar, Kapil*
  - Kapil's slides for MSR-summer school talk.
  - Windows Fabric(Cover slide design)