Specifying Concurrent Problems: Beyond Linearizability and up to Tasks

Sergio Rajsbaum

Joint work with

Armando Castañeda and Michel Raynal
UNAM, Mexico U. Rennes, France

Presented in DISC 2015

extensions in NETYS 2017

Distributed computer scientists excel
at thinking **concurrently**,
and building large distributed systems

Distributed computer scientists excel
at thinking concurrently,
and **building** large distributed systems

Yet, they evade thinking about concurrent problem **specifications**.

Weaver Ants Building Nest from Mango Leaves, Ubon Ratchathani, Thailand

It is infinitely easier and more intuitive for us humans to specify how abstract data structures behave in a sequential setting.

Nir Shavit, CACM 2011

An object

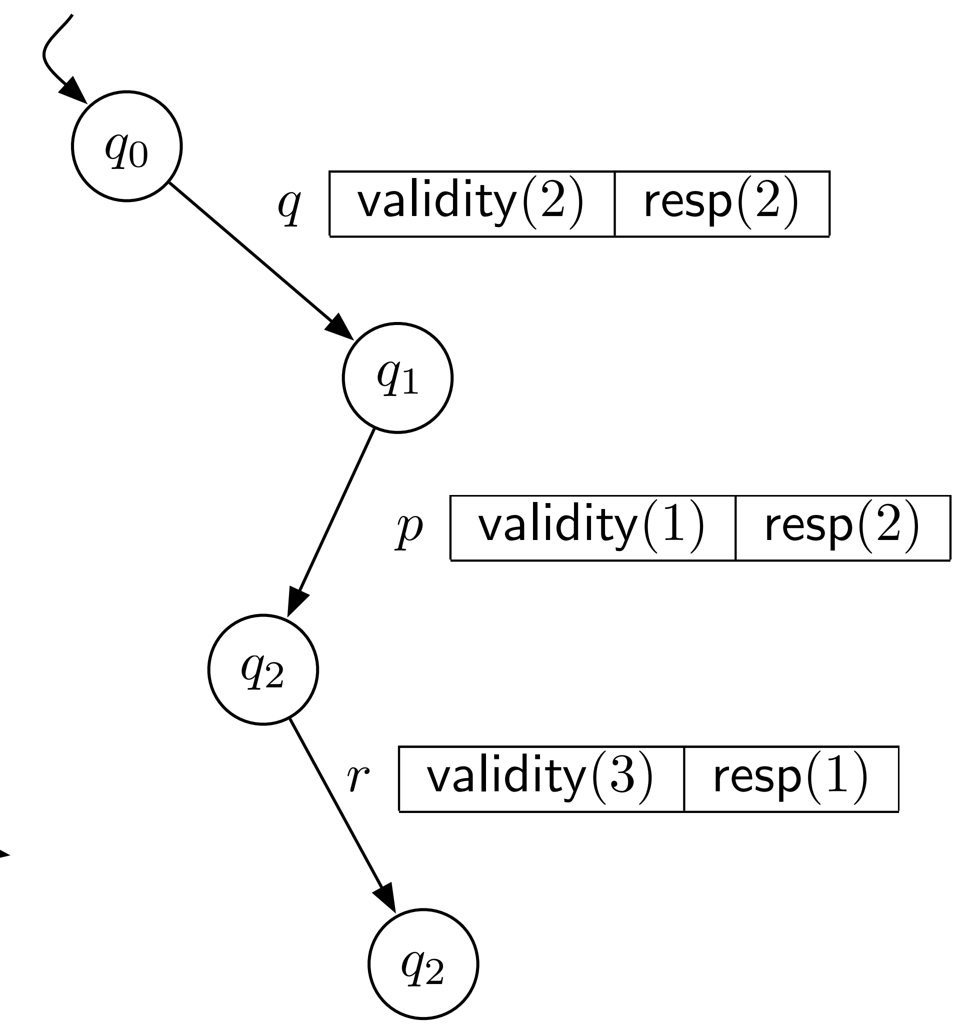
- A central paradigm
- The processes may access it concurrently but specified in terms of a sequential specification, namely…

An object

- an **automaton** describing the outputs the object produces when it is accessed sequentially.
  - Mealy state machine, with transitions of the form

$$\delta(q, in) = (q', r)$$

Example: validity



- Invocations propose input
- responses return values that have been proposed

Sequential specifications are convenient

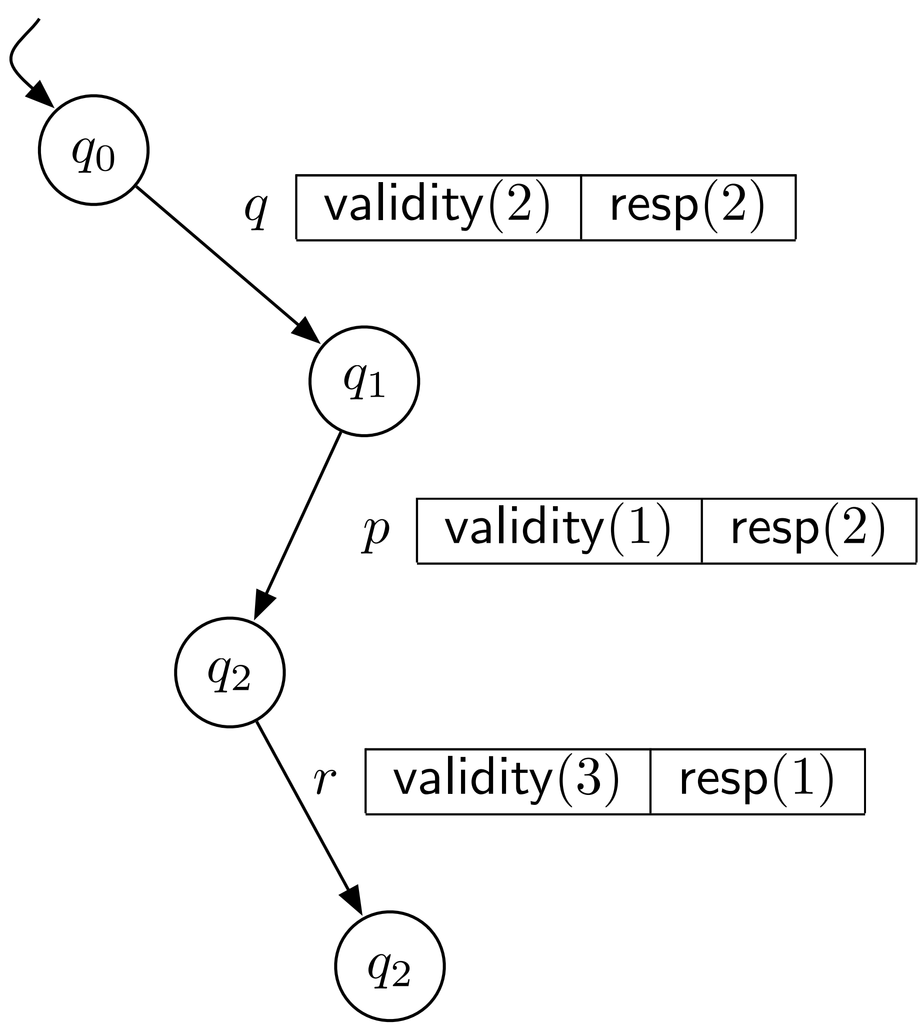
- The paradigm of a sequentially specified object is very convenient:
    - It provides the notion of a state
    - Specification manual grows linearly with the number of operations

Is an implementation correct?

- Given that an object specifies its behaviour only in sequential executions,
- A **correctness** implementation notion is needed for concurrent executions

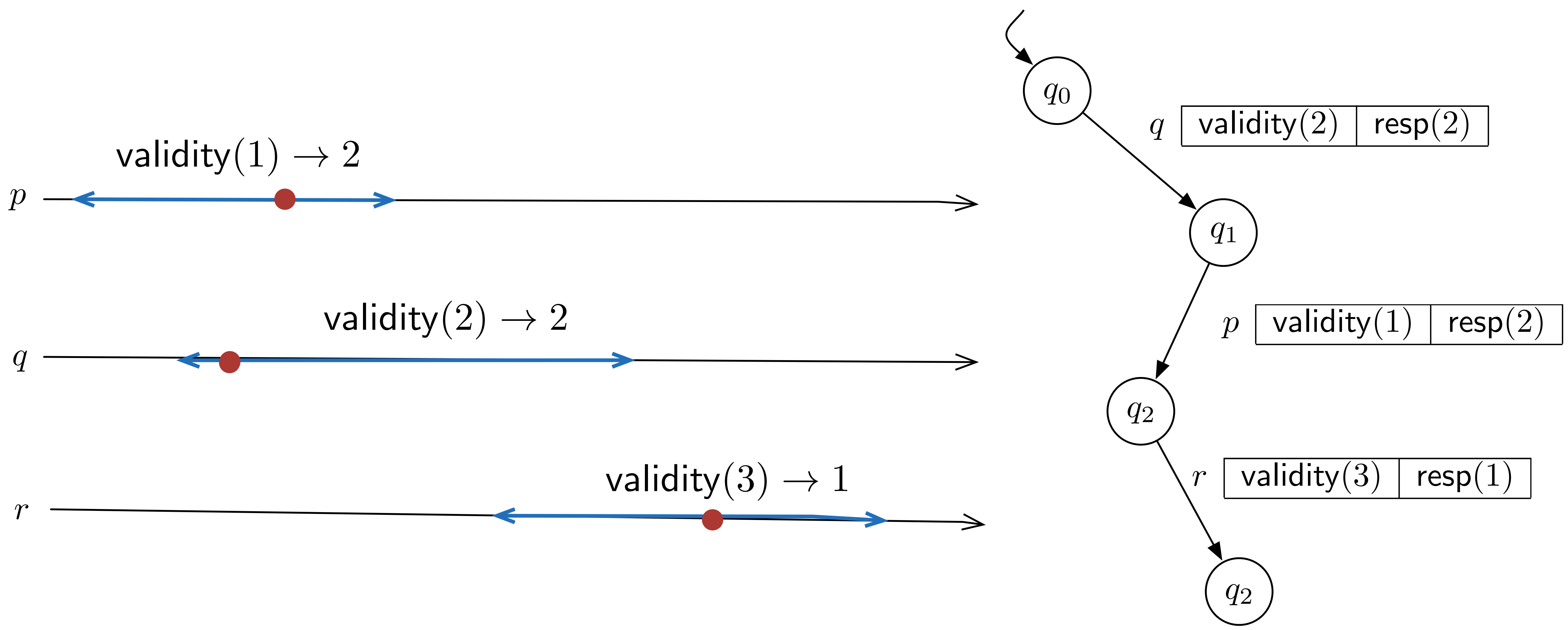
Is an implementation correct?

- Given that an object specifies its behaviour only in sequential executions,
- A **correctness** implementation notion is needed for concurrent executions



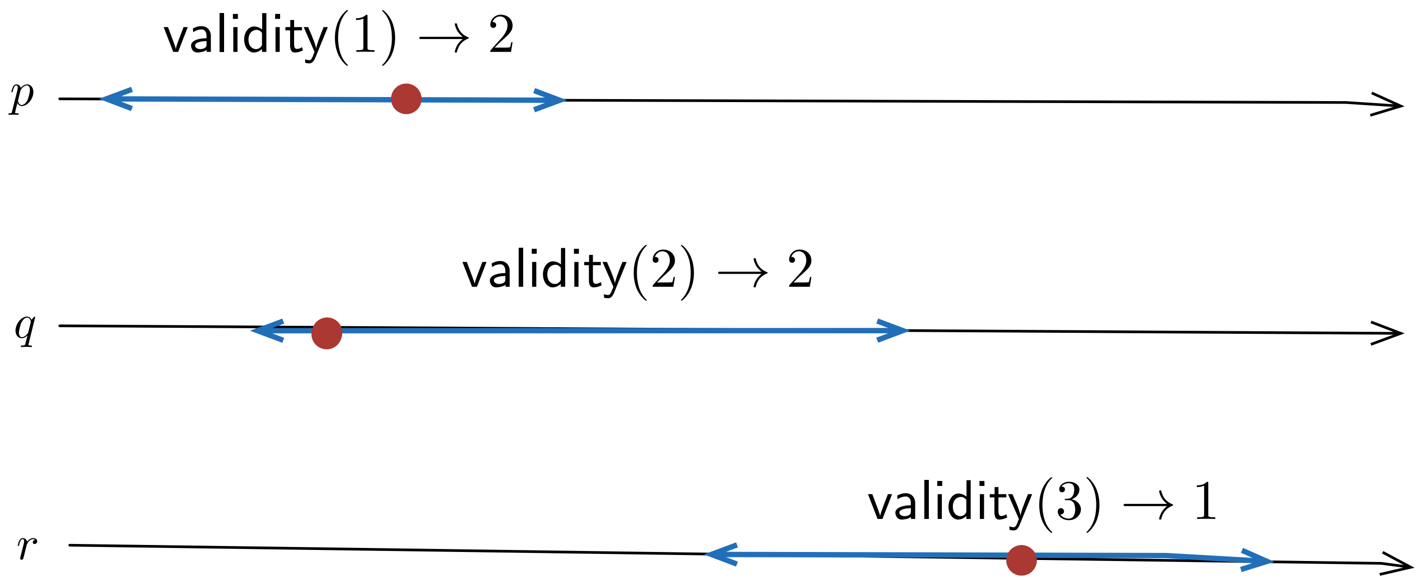
Is an implementation correct?

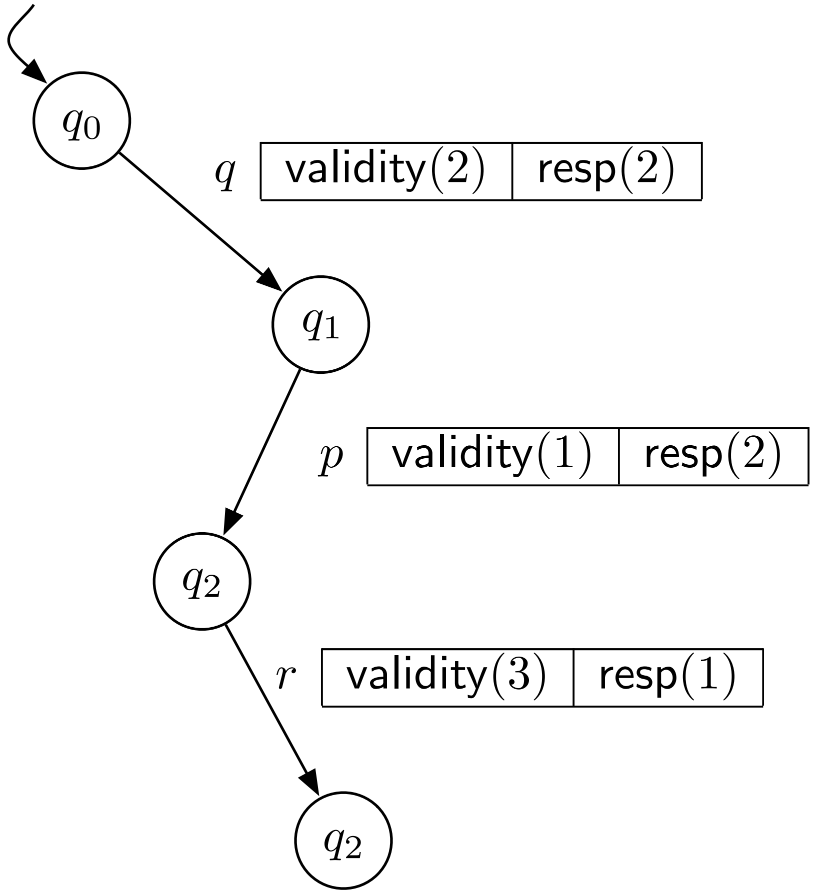
- Given that an object specifies its behaviour only in sequential executions,
- A **correctness** implementation notion is needed for concurrent executions



Linearizability

- Operations seem to occur at a point, in between invocation and response,
- i.e., they can be transformed to a valid sequential execution.



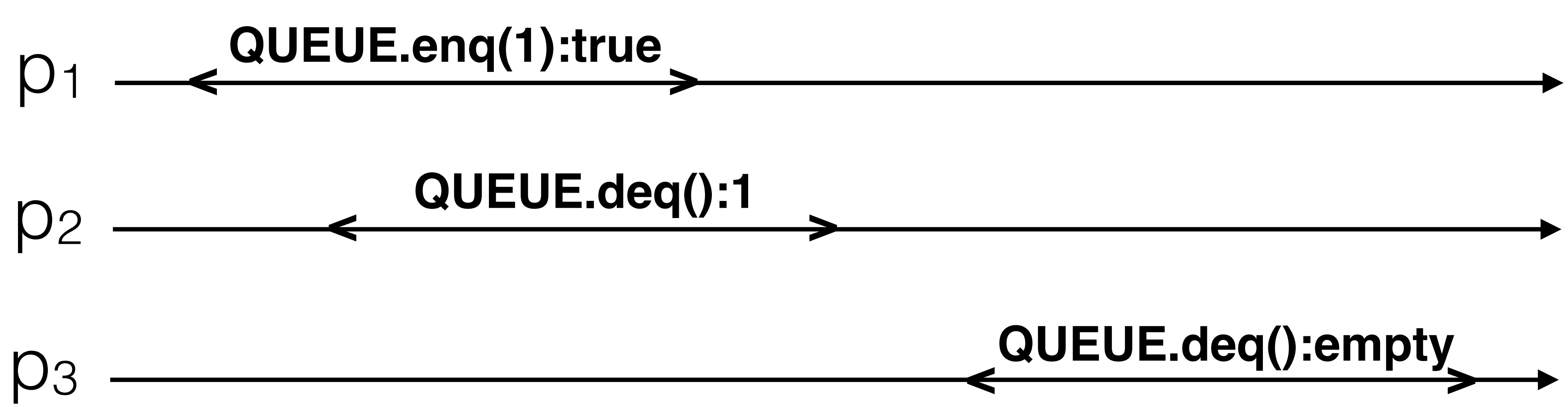


Queue

- Often concurrent objects come from sequential world.
- Operations seem to occur sequentially, i.e., they can be **transformed** to a **valid sequential execution.**

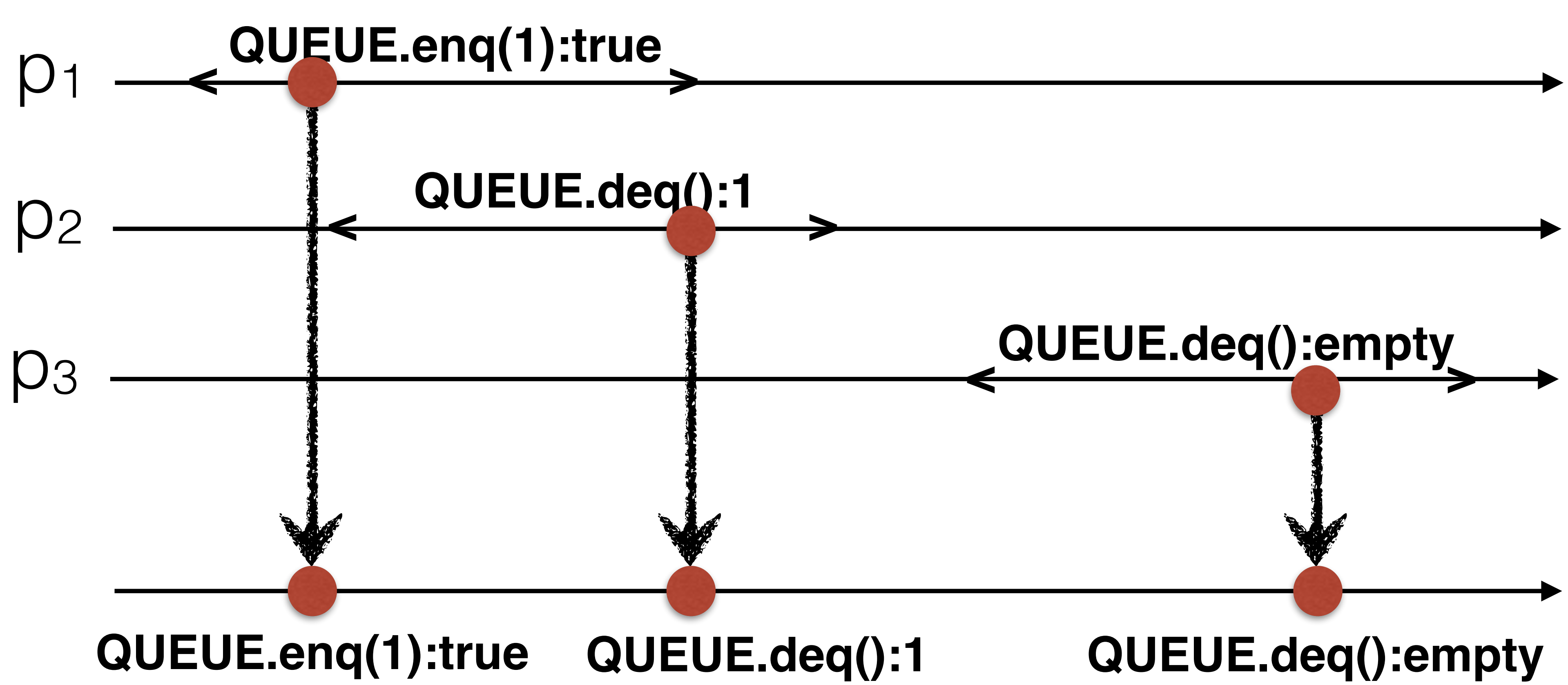
Queue

- Often concurrent objects come from sequential world.
- Operations seem to occur sequentially, i.e., they can be **transformed** to a **valid sequential execution.**



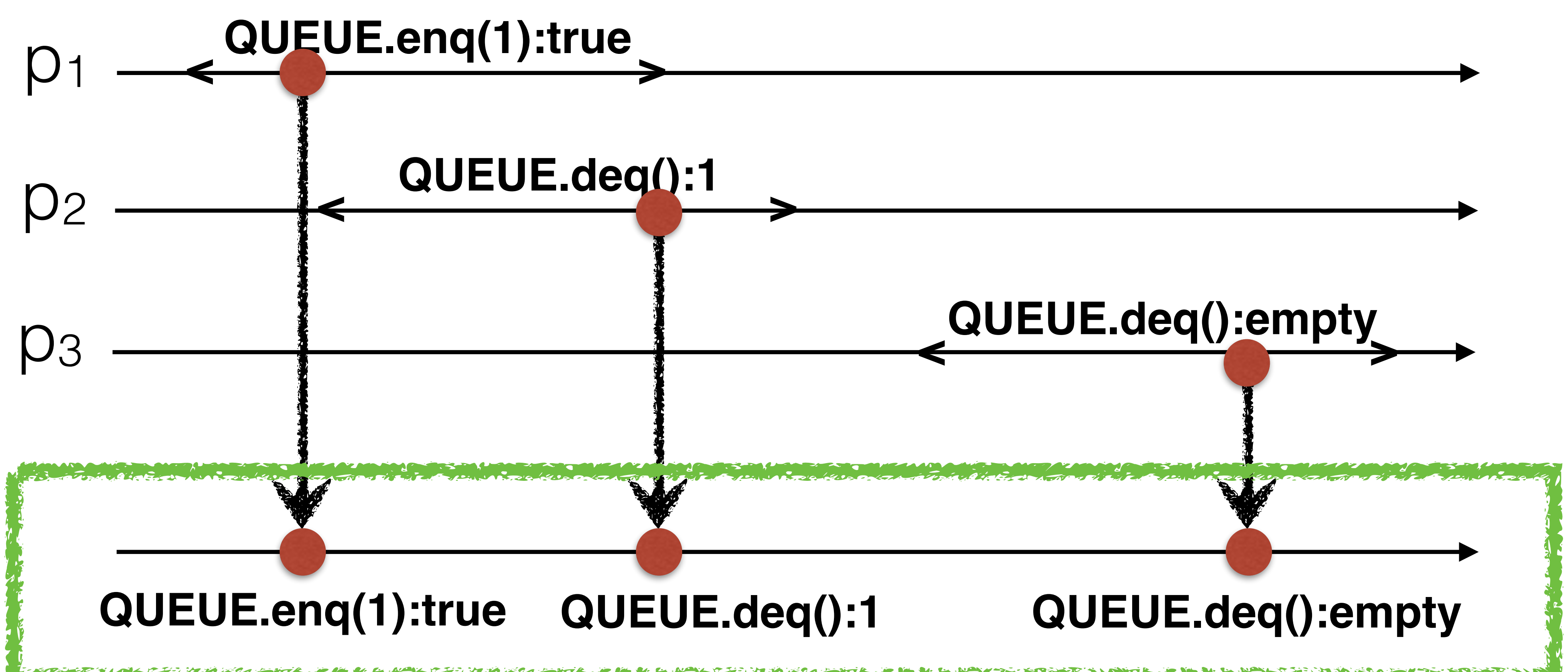
Queue

- Often concurrent objects come from sequential world.
- Operations seem to occur sequentially, i.e., they can be **transformed** to a **valid sequential execution.**



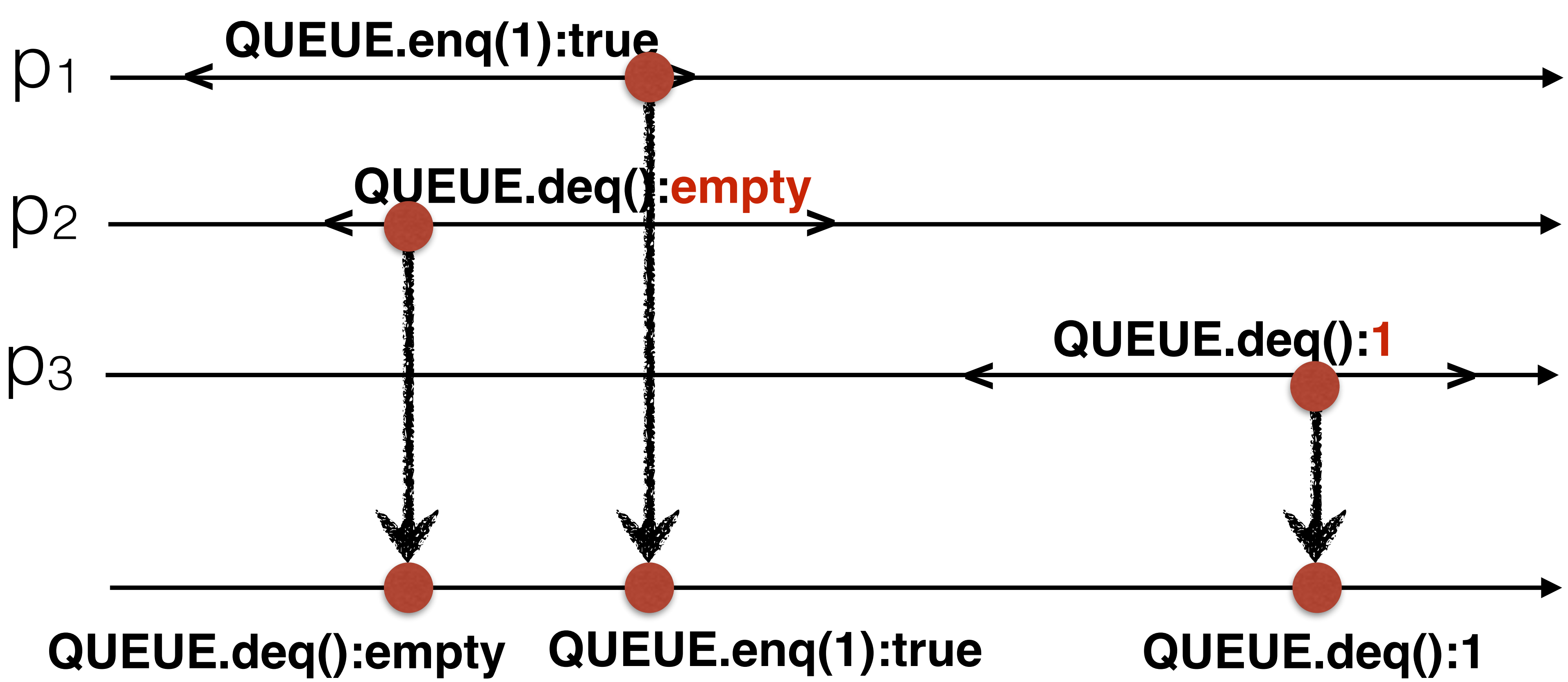
Queue

- Often concurrent objects come from sequential world.
- Operations seem to occur sequentially, i  an be **transformed** to a **valid sequential ex**



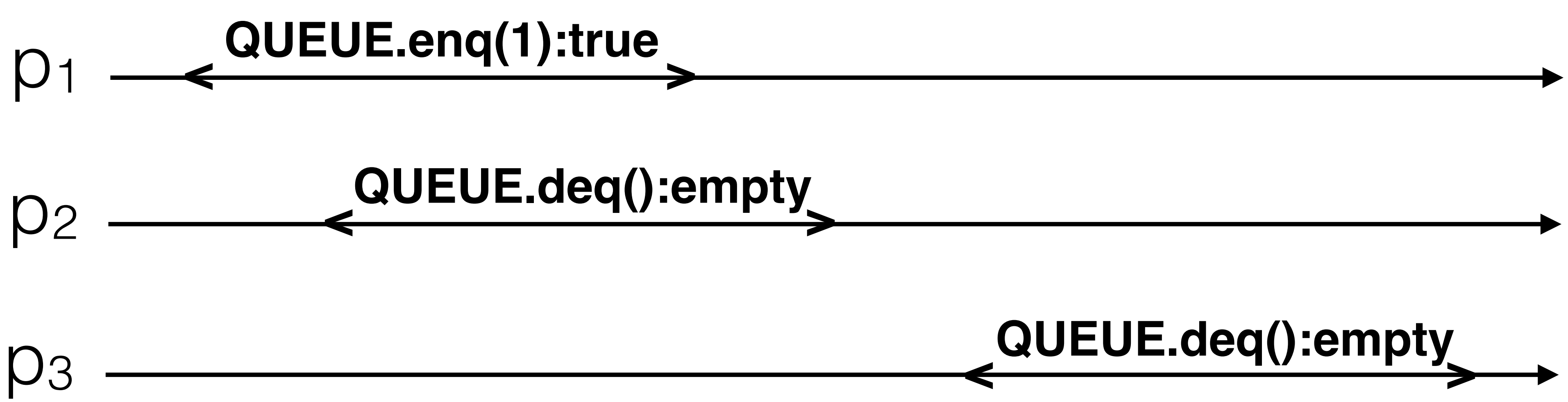
Queue

- Standard correctness criteria.
- **Linearizability:** Operations seem to occur sequentially, i.e., they can be **transform** **alid** sequential execution.



Queue

- Standard correctness criteria.
- **Linearizability:** Operations seem to occur sequentially, i.e., they can be **transforme** **lid** sequential execution.



Importance of Linearizability

- Clear specifications. Easy to think sequentially.
- **Good properties** for the development of systems:

  Non-blocking: It never forces the system to block

  Locality: Modular approach. Linearizable implementations compose a linearizable system.

Importance of Linearizability

- Clear specifications. Easy to think sequentially.
- **Good properties** for the development of systems:

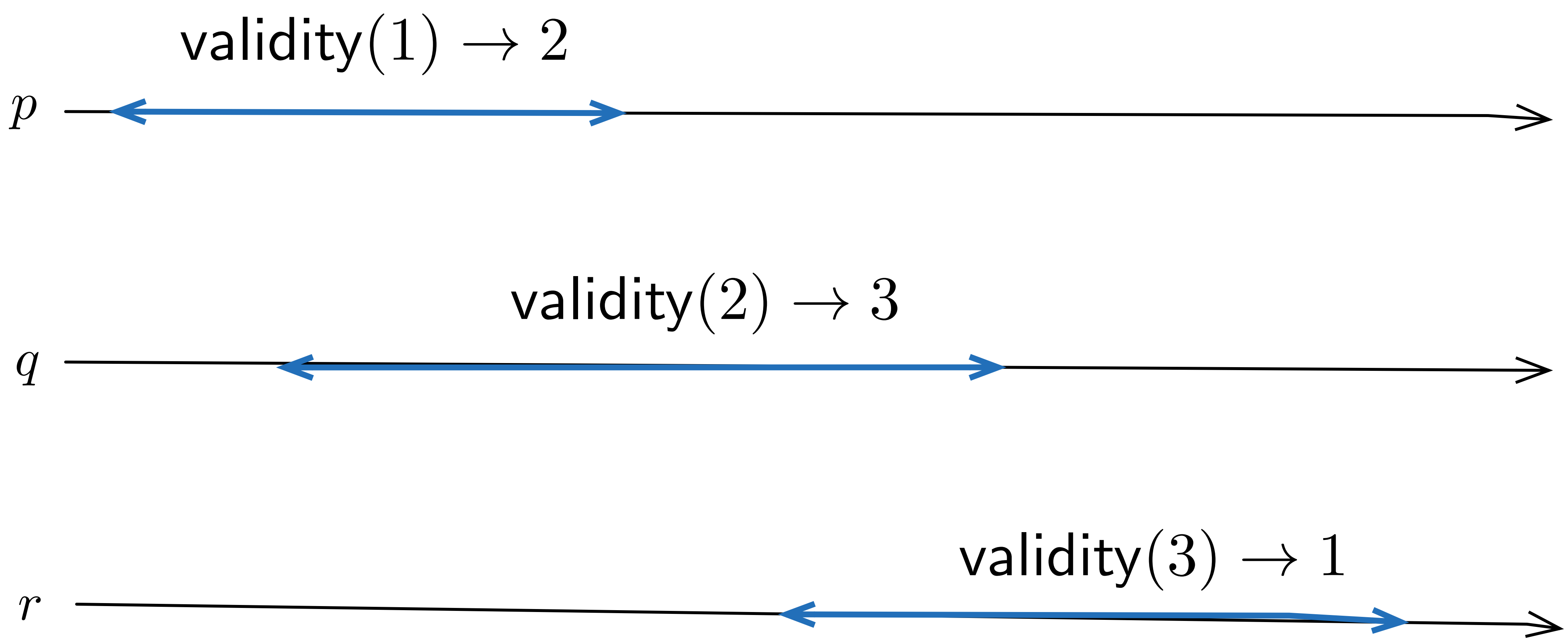
There are limitations!!

Locality: Modular approach. Linearizable implementations compose a linearizable system.

Distributed object

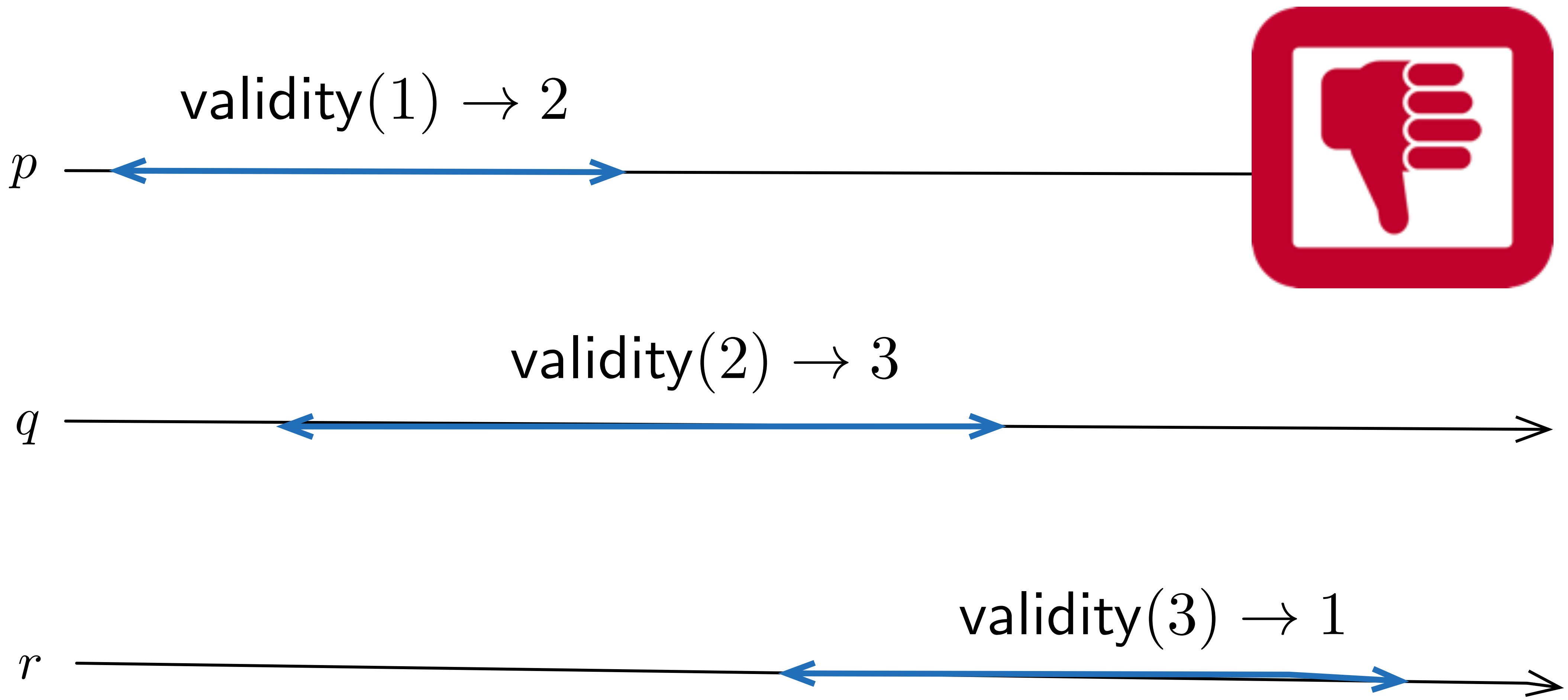
- Are all distributed problems objects?
- No!
- What *is* a distributed object?

Validity object



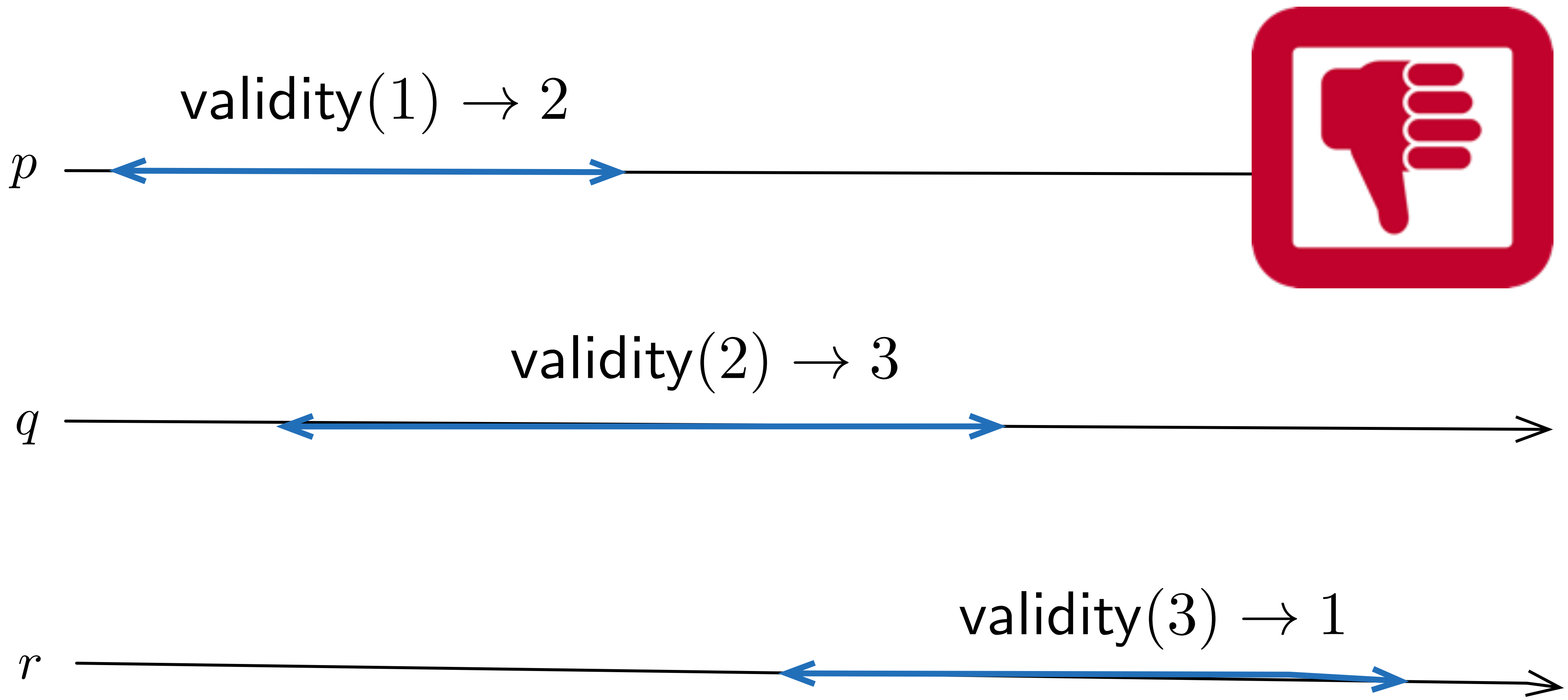
- There is a simple implementation based on read/write primitives

Validity object



- There is a simple implementation based on read/write primitives

Validity object



- Ther d on
  read

not linearizable

Snapshot Object

- Shared memory M; one entry per process

  write(i, v): atomically writes v in M[i]

  snapshot(M): takes an atomic snapshot of M

- Has a natural sequential specification

- Several **linearizable** implementations based on **read/write** primitives

Write-Snapshot Object

- In some applications a **snapshot** always goes after a **write**
- New object with a single operation
- **write-snapshot(i, v)**: writes v in M[i] and takes a snapshot of the memory.
- Let’s focus on **one-shot** for this talk
- How do we specify it?

Informal specifications

- **write-snapshot(v)**: writes and takes a snapshot of the memory

- Usual property-based specification:
  1. **Self-inclusion**: each S_i contains i
  2. **Containment**: every S_i, S_j are comparable under containment
  3. **Validity**: if j is S_i in j was written in M[j]

Concurrent-based specifications

- Used in distributed computability (often using topology)
- Main example: k-set agreement and consensus
- Many others, loop agreement, adopt-commit, renaming, etc.
- **propose(x)**: each process has an input x, returns a value y
- Usual property-based specification for k-set agreement:
  1. **Agreement**: at most k different values are returned
  2. **Validity**: an output value y was proposed

More formal: Tasks

- **One-shot** distributed problem
- **Static** approach
- **Task** :
  1. **Input configurations** (simplicial complex)
  2. **Output configurations** (simplicial complex)
  3. **Input/output relation**
- Less explored but fundamental: computability, topological approach, simulations

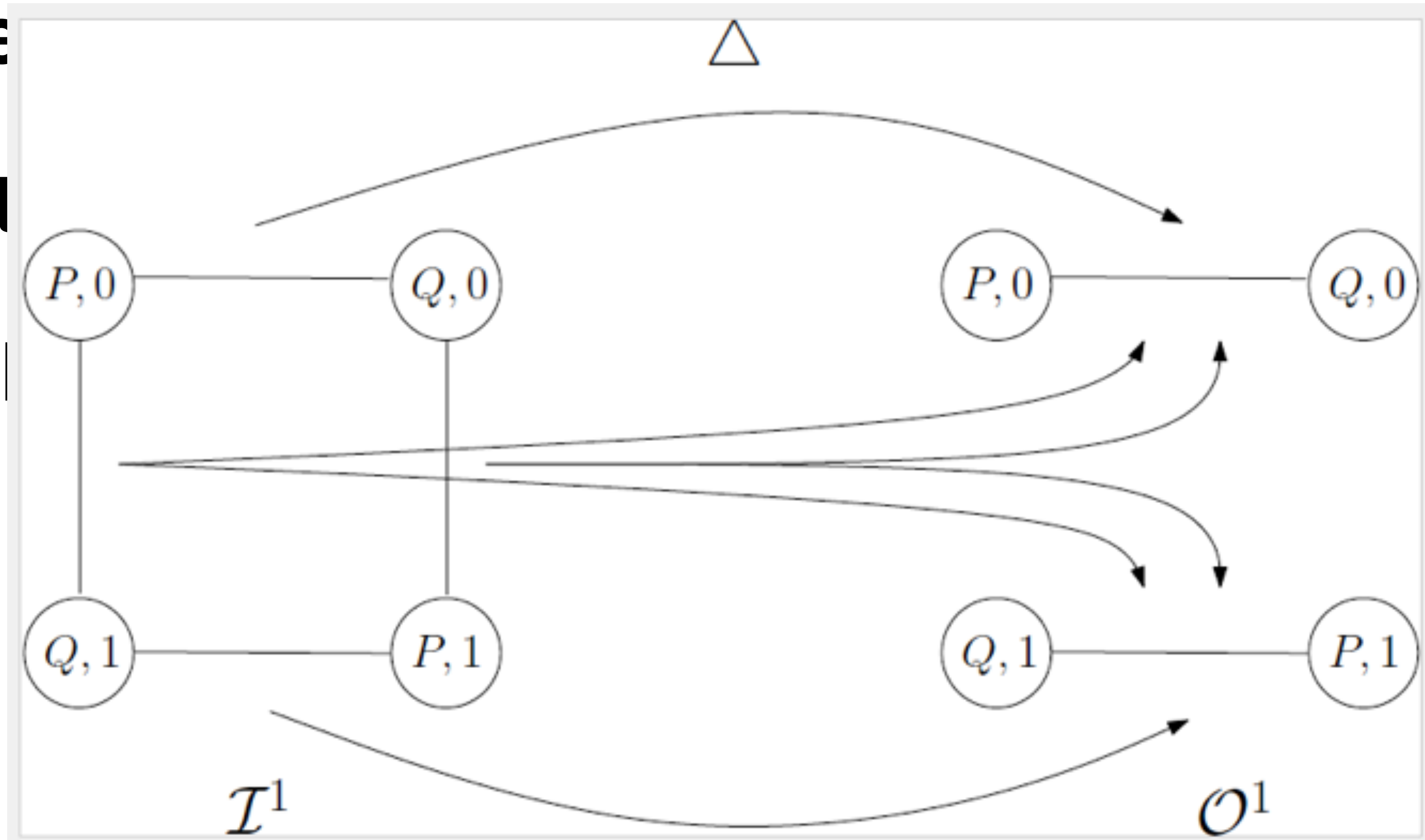
More formal: Tasks

- **One-shot** distributed problem
- **Static** approach
- **Task** :
    1. **Input configurations** (simplicial complex)
    2. **Output configurations** (simplicial complex)
    3. **Input/output relation**
-

Tasks tell what might happen in presence of concurrency

More formal: Tasks

- **One**
- **Stat**
- **Tas**
  1. ex)
  2. plex)
  3.



-

Tasks tell what might happen in presence of concurrency

Solving Tasks

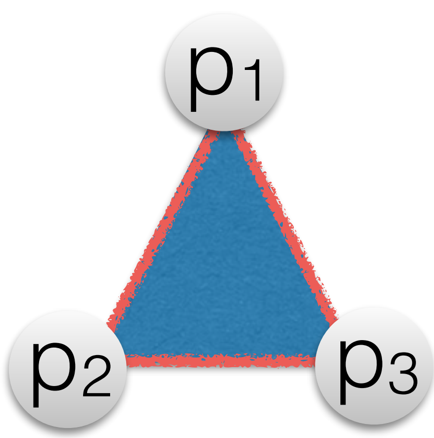
- When does an algorithm **solves** a task ?

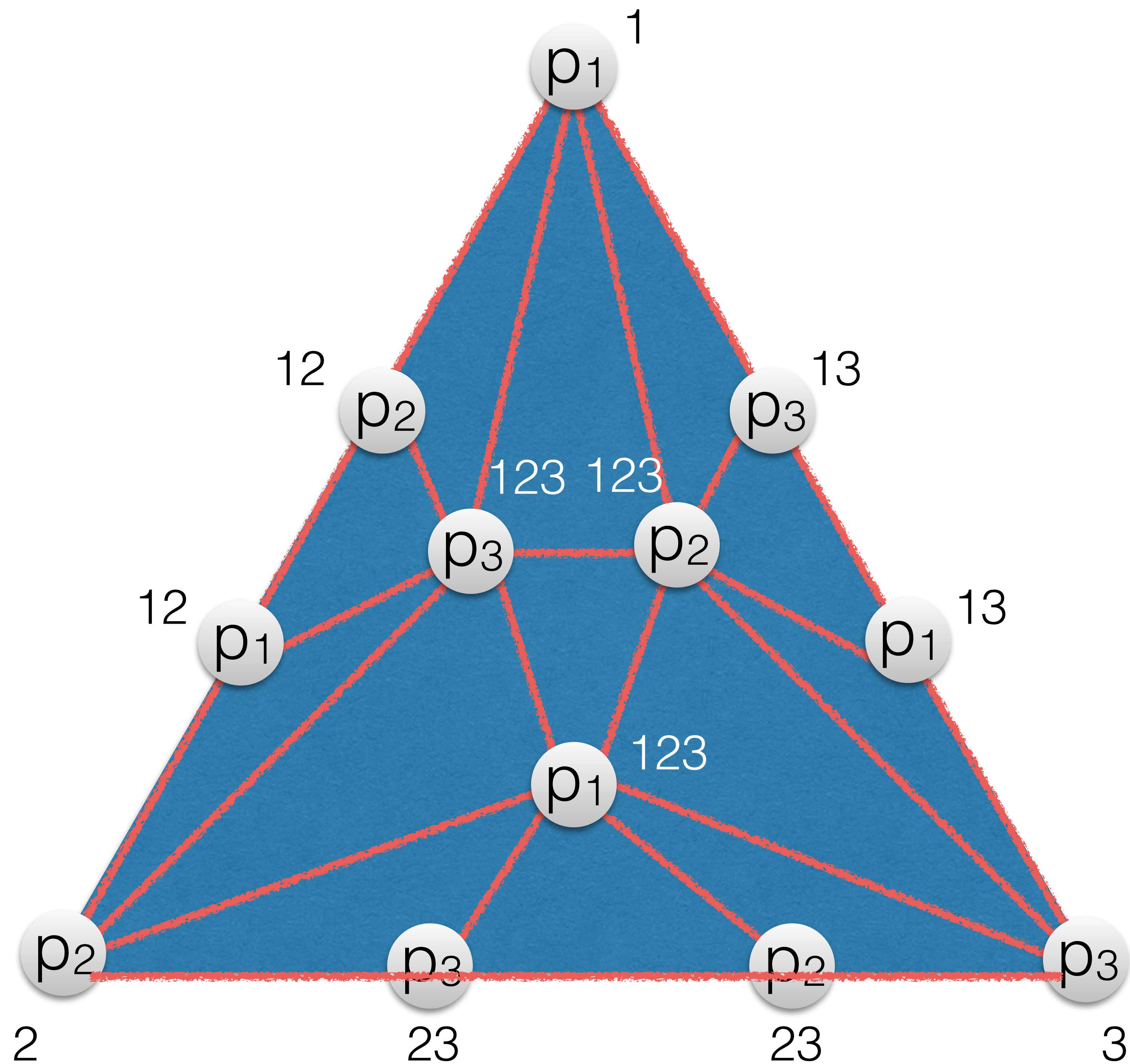
  For each set of participating processes, in every execution, **inputs and outputs** in every execution **agree** with the **mapping** specifying the task

Importance of Tasks

- Basic **computability** unit, distributed equivalent of a function
- Study of **set agreement** and **renaming** lead to a **connection** between **distributed computing** and **topology**
- **but:** Semantic of tasks is not well studied. What are they? Certainly, not sequential objects

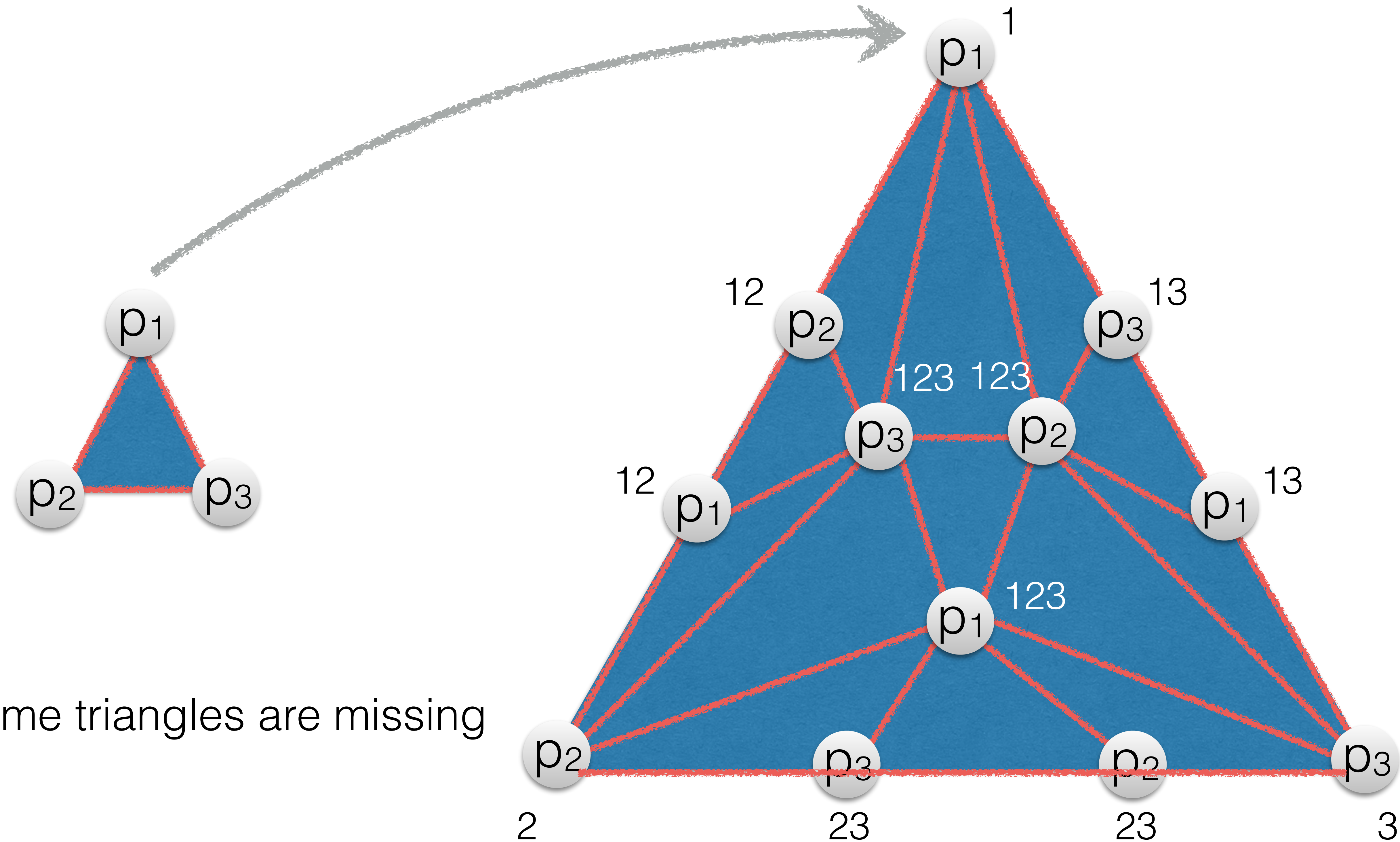
Write-Snapshot Task





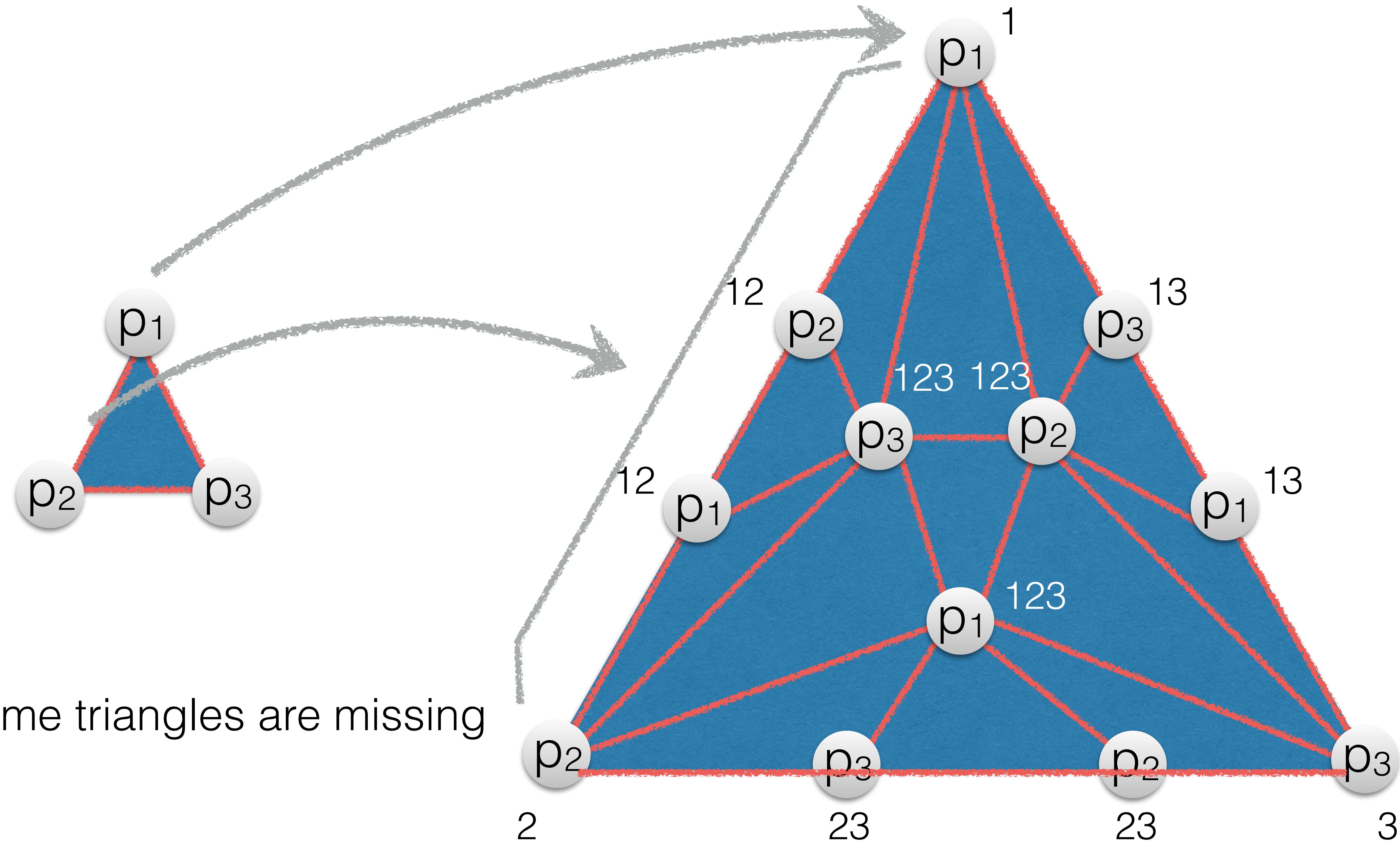
*Some triangles are missing

Write-Snapshot Task



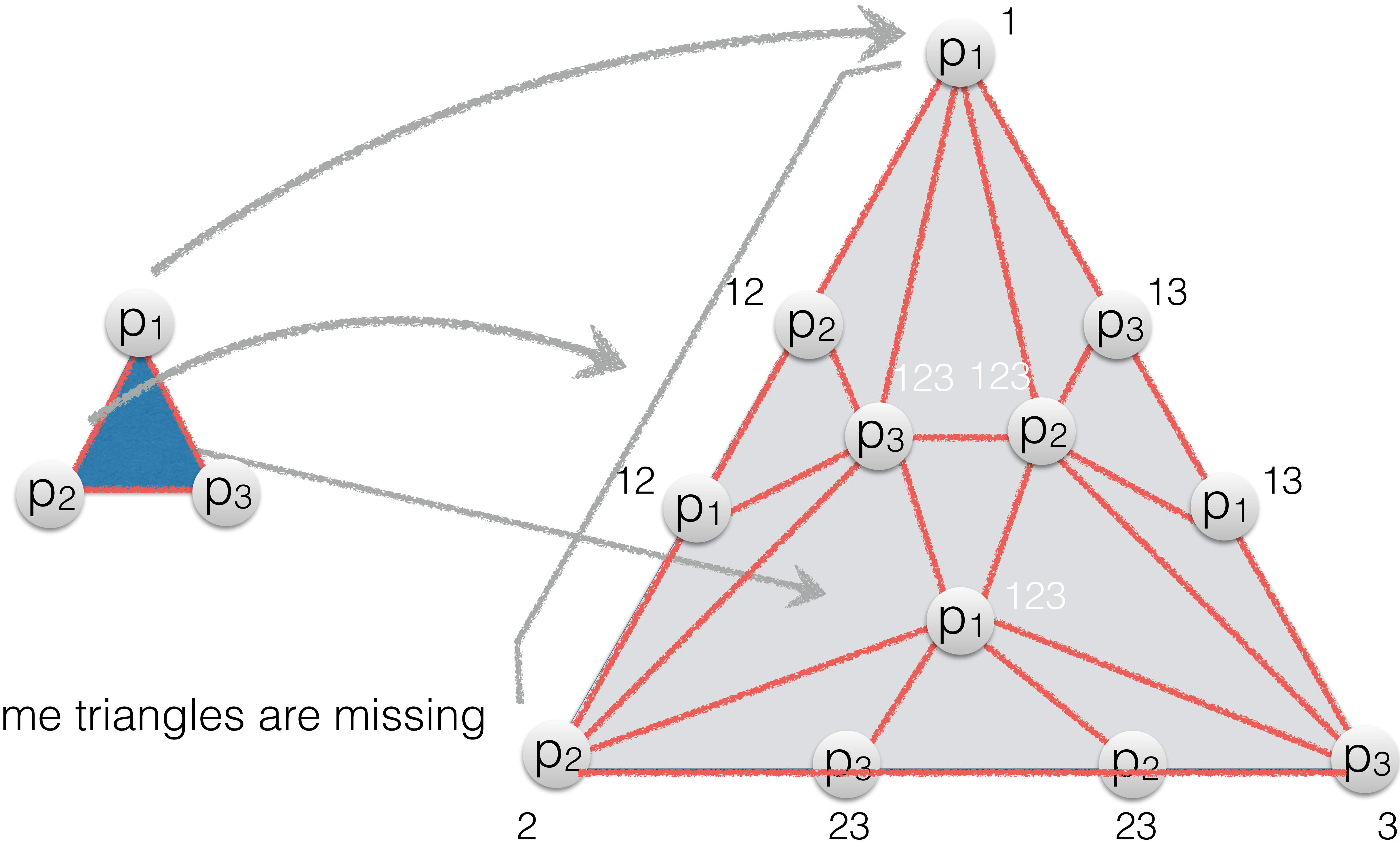
*Some triangles are missing

Write-Snapshot Task



*Some triangles are missing

Write-Snapshot Task



*Some triangles are missing

Write-Snapshot Object

- An implementation based on **read/write**

operation write_snapshot(i) **is** % issued by p_i
$MEM[i] \leftarrow i$;
$new_i \leftarrow \cup_{1 \leq j \leq n}\{MEM[j]$ such that $MEM[j] \neq \bot\}$;
repeat $old_i \leftarrow new_i$;
$new_i \leftarrow \cup_{1 \leq j \leq n}\{MEM[j]$ such that $MEM[j] \neq \bot\}$
until $(old_i = new_i)$ **end repeat**;
return(new_i).

Write-Snapshot Object

- An implementation based on **read/write**

Is it linearizable?
Is there a sequential specification?

repeat $old_i \leftarrow new_i$;
$\quad new_i \leftarrow \cup_{1 \leq j \leq n}\{MEM[j] \text{ such that } MEM[j] \neq \bot\}$
until $(old_i = new_i)$ **end repeat**;
return(new_i).

Write-Snapshot Object

- An implementation based on **read/write**

Is it linearizable?

Is there a sequential sp

NO!!

(04) $new_i \leftarrow \cup_{1 \leq j \leq n}\{MEM[j]$ such that $MEM[j] \neq \perp\}$

(05) **until** $(old_i = new_i)$ **end repeat**;

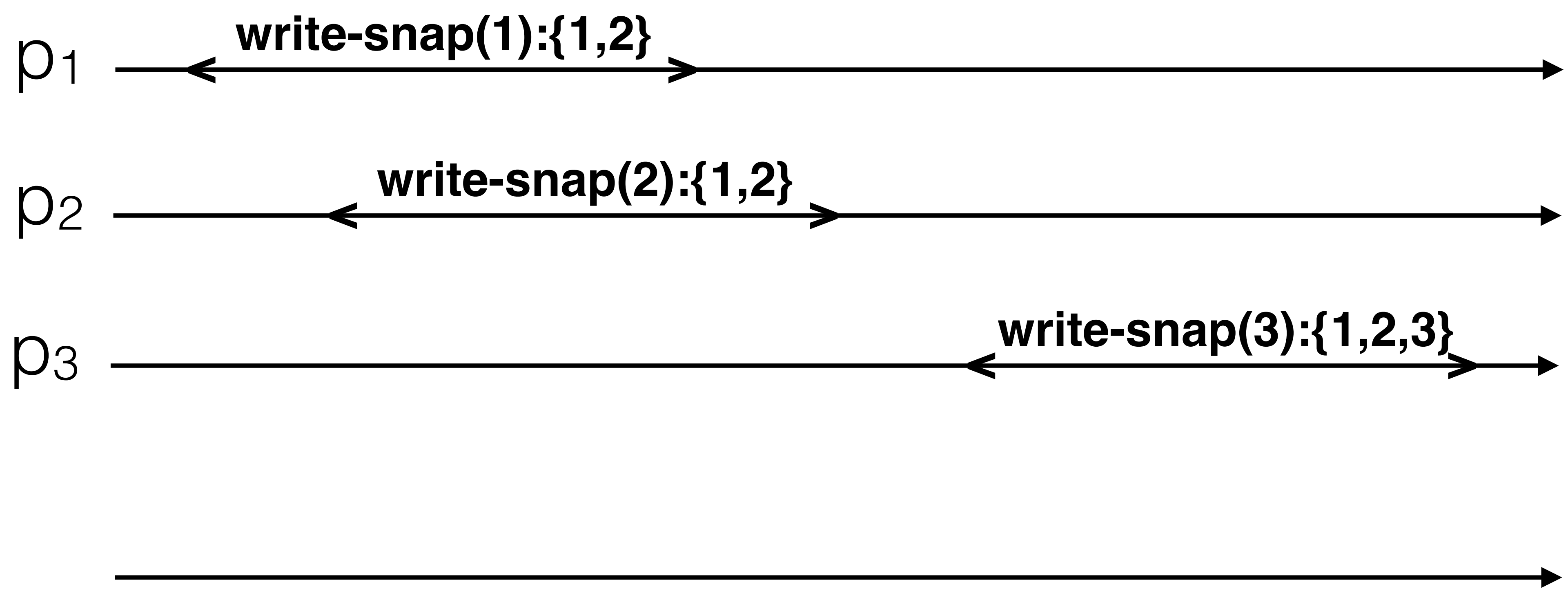
(06) return(new_i).

Write-Snapshot Object

- There is no sequential specification
- If there is such an specification, in each execution of a **read/write** linearizable implementation, there is a **‘first’** process
- Solve **Test&Set** from any such **read/write** implementation. A contradiction!!
- What is going on?

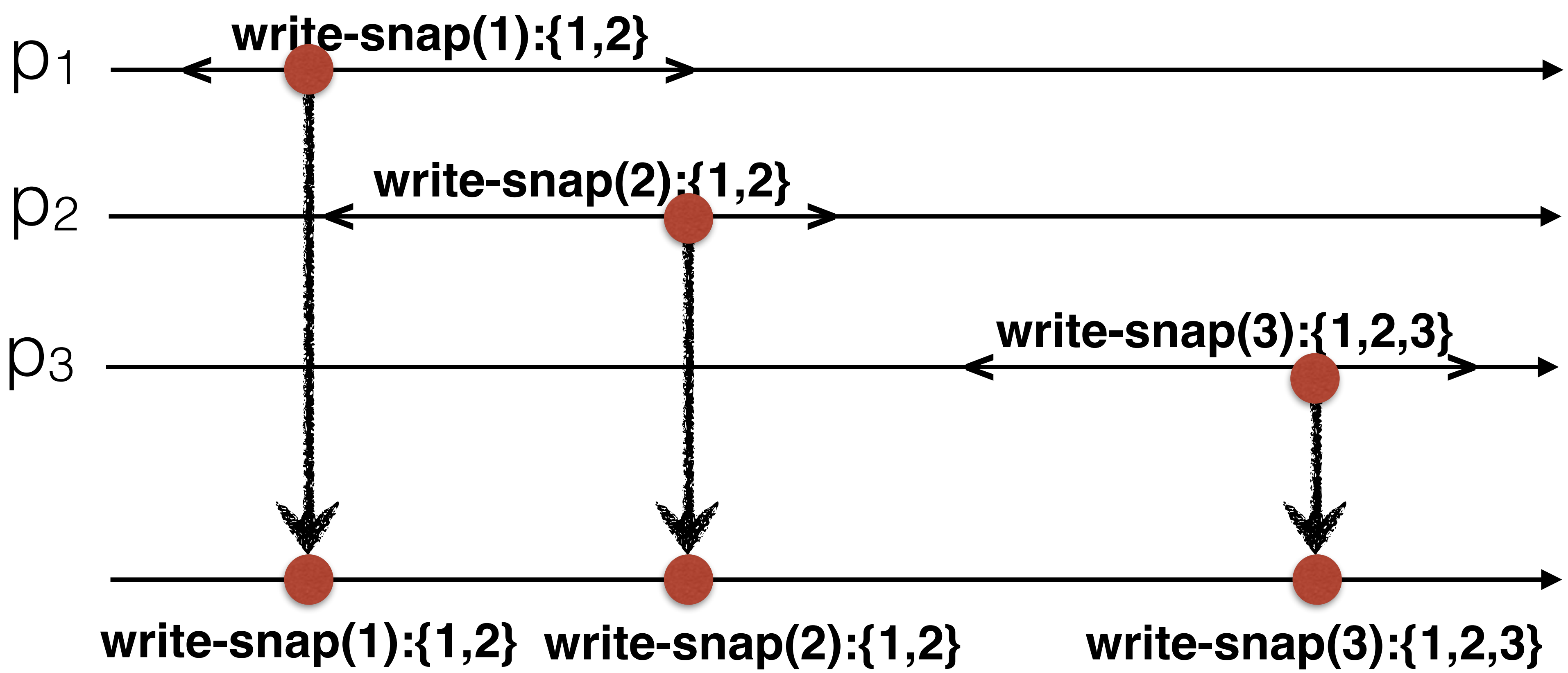
Write-Snapshot Object

- *Tasks can model executions that sequential specs cannot:*



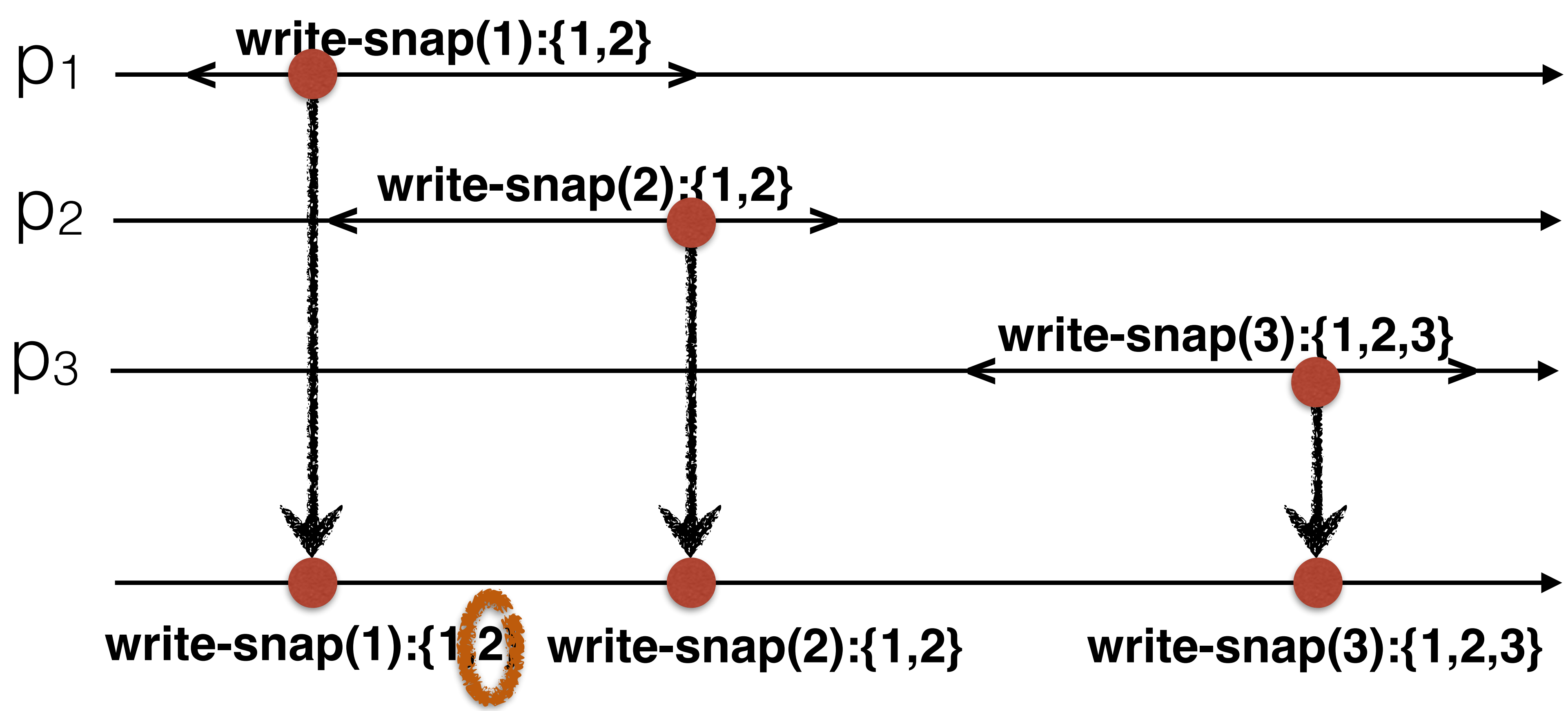
Write-Snapshot Object

- *Tasks can model executions that sequential specs cannot:*



Write-Snapshot Object

- *Tasks can model executions that sequential specs cannot:*



Write-Snapshot Object

- Any sequential spec. of **write-snapshot** models a **proper subset of executions**

- The **resulting specification** is **stronger** than the object we want to model

Limitations of Linearizability

- First noted by Neiger BA PODC'94: **NO sequential specification** for **set agreement** and **immediate snapshot** (property-based specification)
- **Set linearizability**
- Similar approach: **concurrency-aware** by Hemed, Rinetzky and Vafeiadis DISC'15
- **Not enough** to specify **write-snapshot**

Examples of non-sequentially specifiable tasks:

1. Adopt-commit (used in Paxos for safety)
2. Conflict-detection (Aspnes-Ellen)
3. Safe-consensus (weaker validity of consensus)
4. Immediate snapshot (Asyn. Computability Theorem)
5. k-set agreement (generalization of consensus)
6. Exchanger (Java object)

Limitations of Tasks

- A **one-shot queue** (or **stack**) **cannot** be **specified** as a **task**.

- **Problem:** Tasks have no mechanism to model memory of automatons

Limitations of Tasks

- A **one-shot queue** (or **stack**) **cannot** be **specified** as a **task**.

Linearizability and Tasks are importan but not unified!!

Limitations of Tasks

- A **one-shot queue** (or **stack**) **cannot** be **specified** as a **task**.

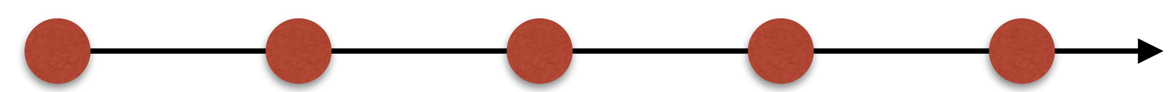
Linearizability and Tasks are importan but not unified!!

Our contribution: Unify these two styles of specifications

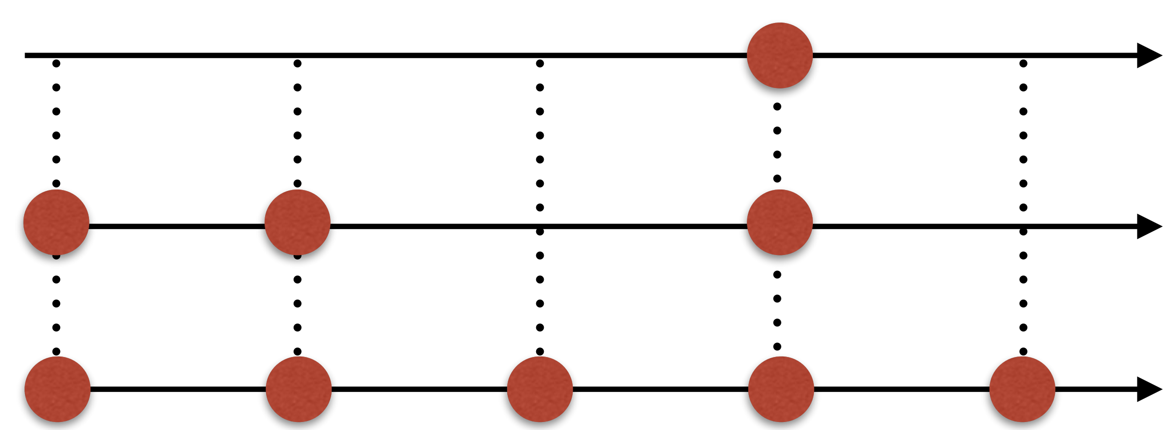
Set Linearizability (Neiger 94)

- Go from dimension 1 to dimension 2:

Sequential

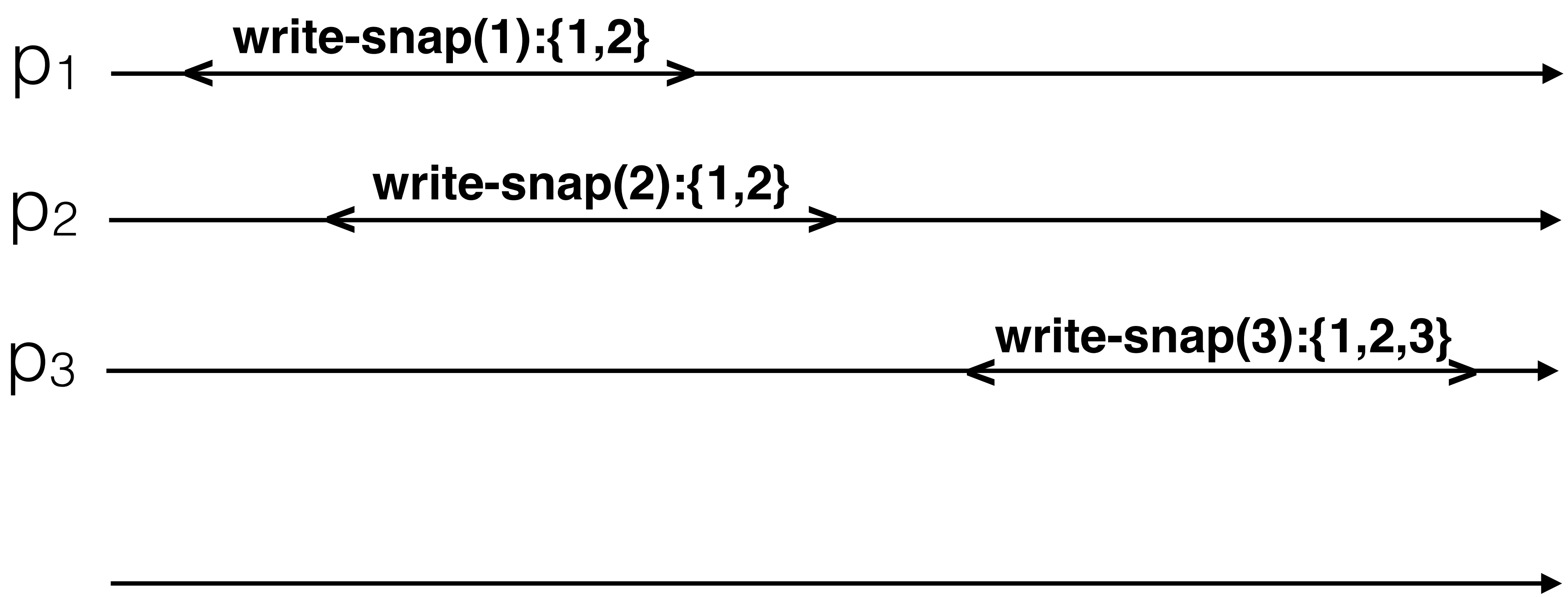


Set sequential



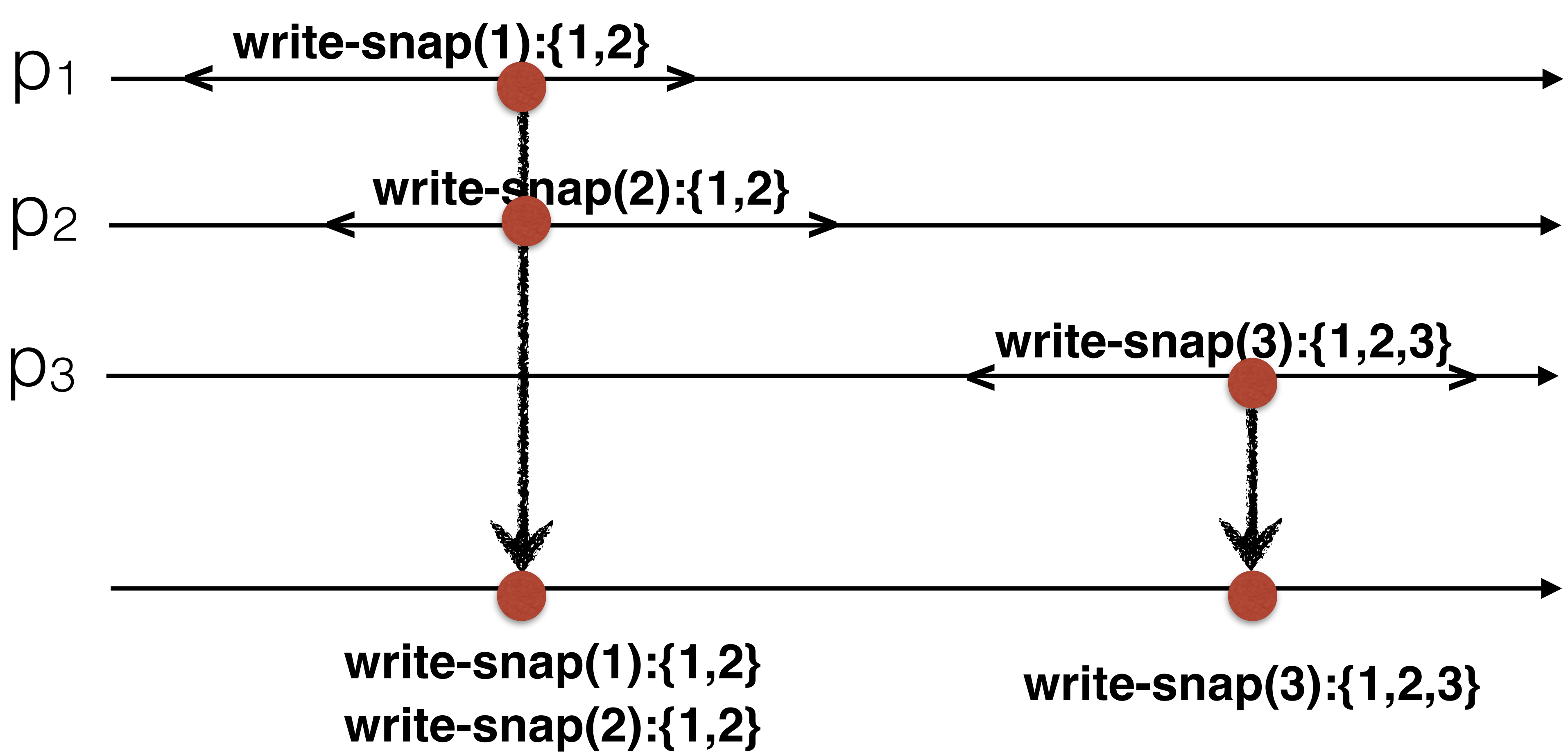
Set Linearizability (Neiger 94)

- Go from dimension 1 to dimension 2:



Set Linearizability (Neiger 94)

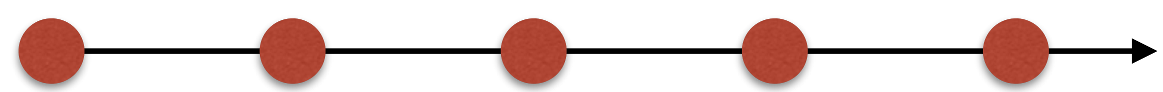
- Go from dimension 1 to dimension 2:



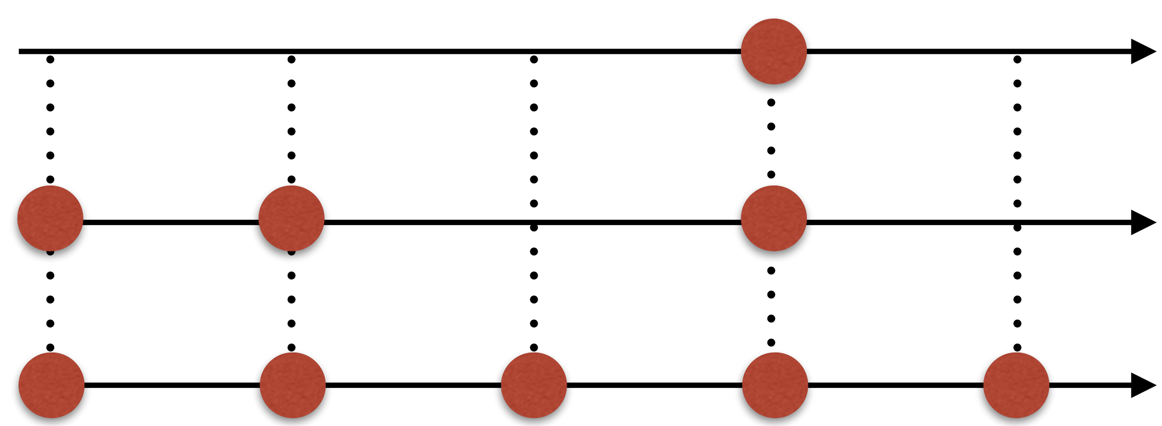
Set sequential automata (Neiger 94)

- Transitions labeled with *sets* of operations and their responses

Sequential

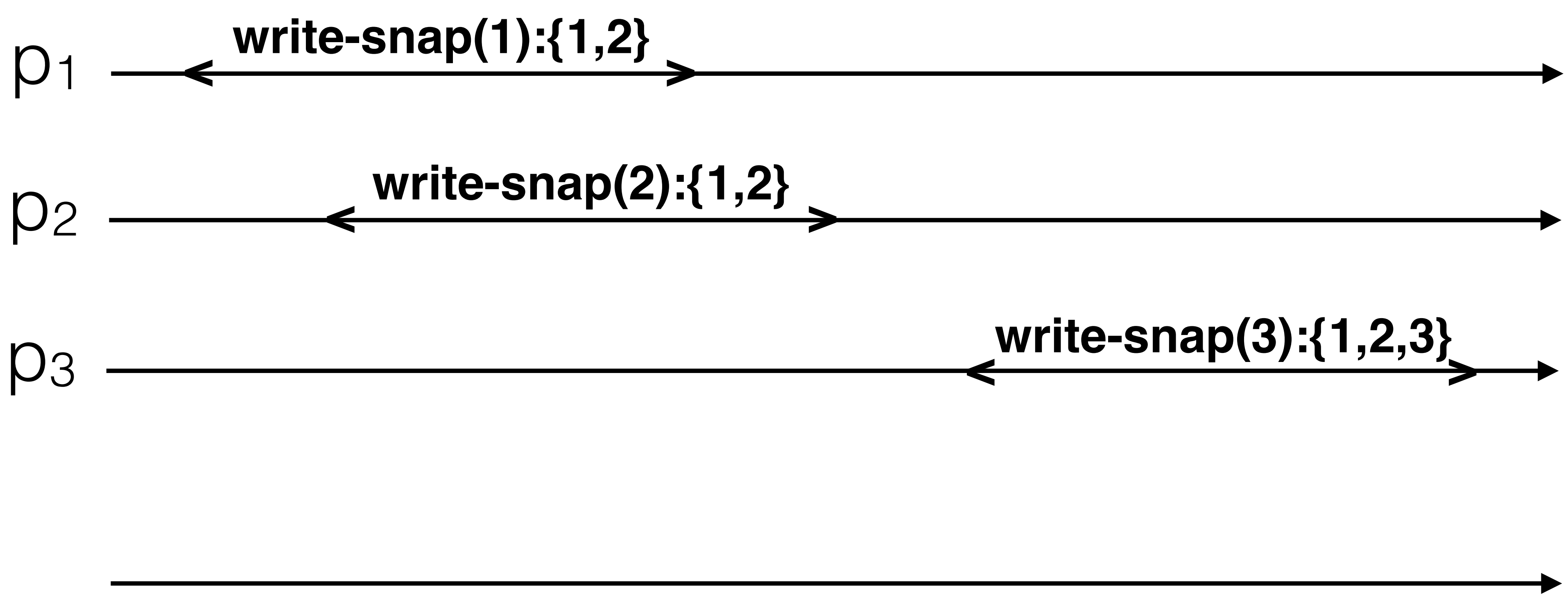


Set sequential



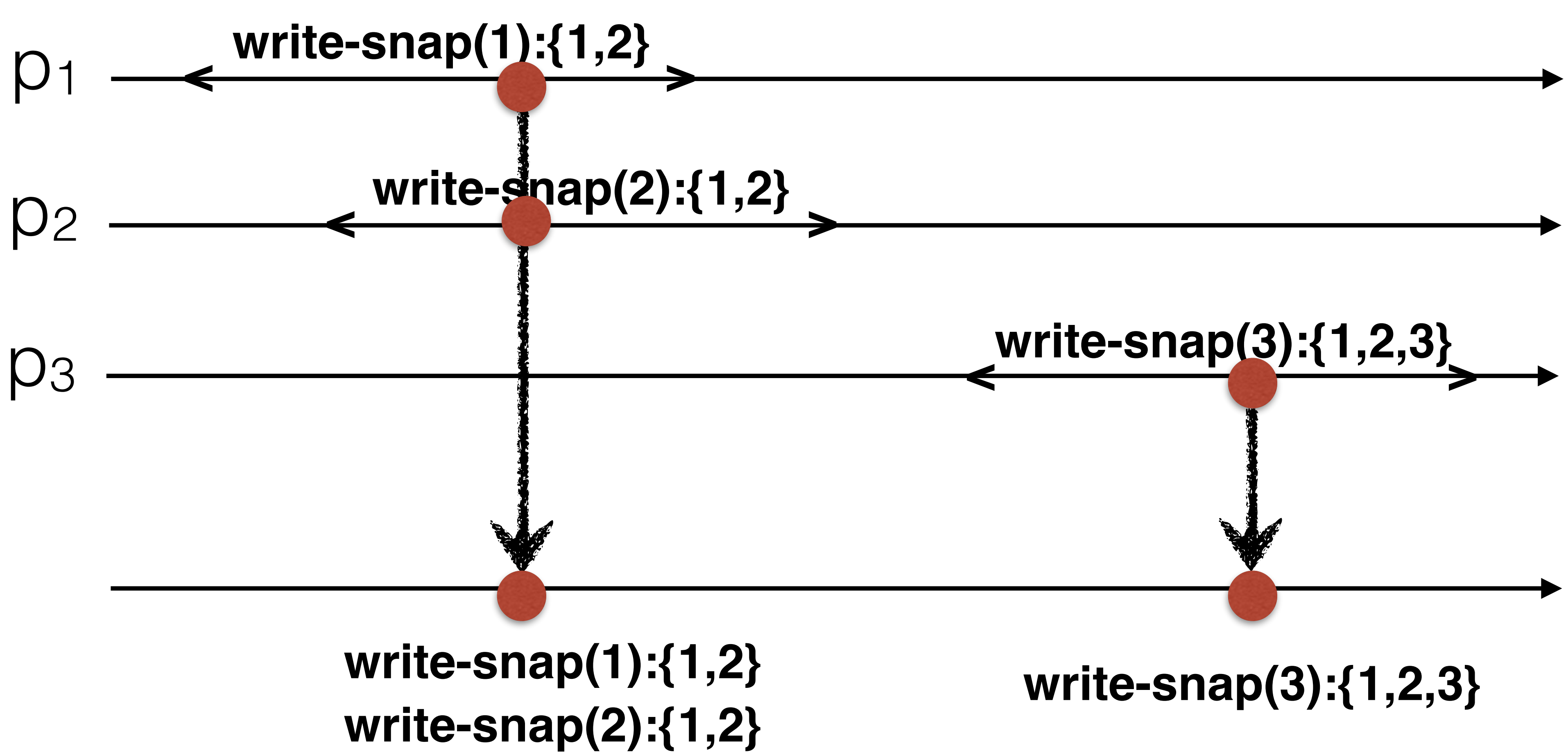
Set sequential automata (Neiger 94)

- Transitions labeled with *sets* of operations and their responses



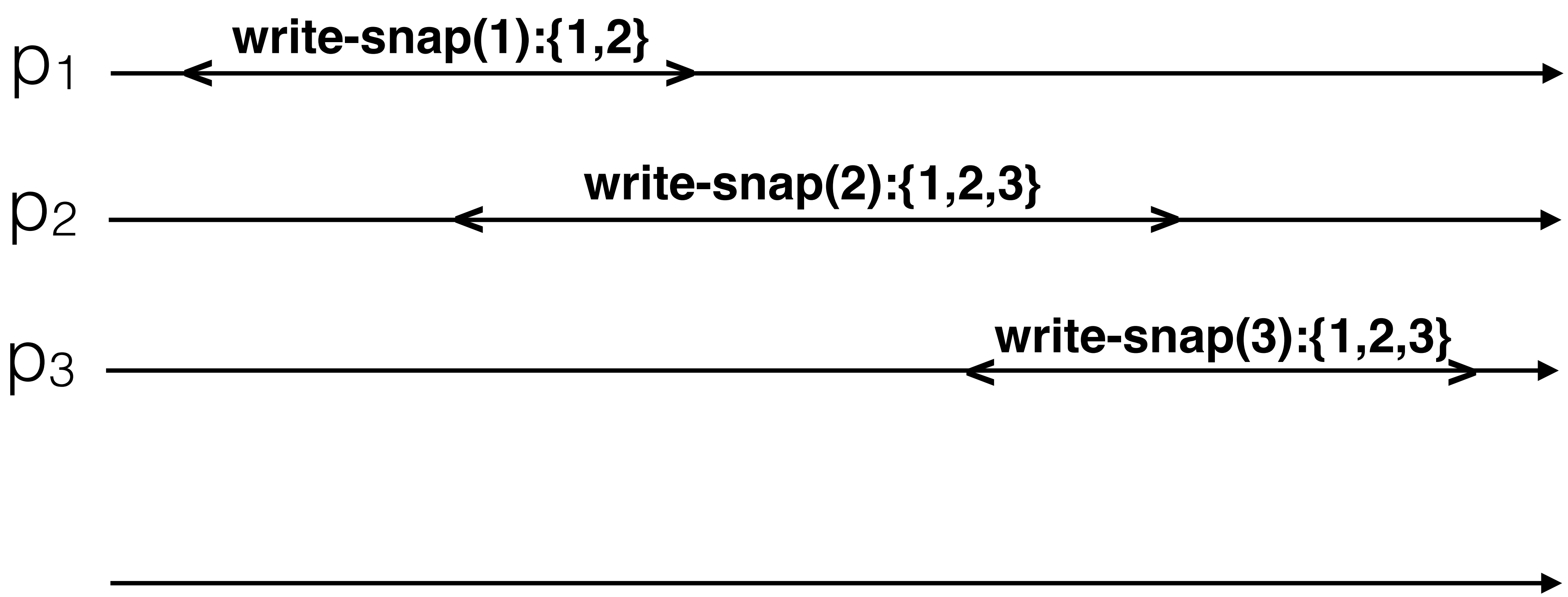
Set sequential automata (Neiger 94)

- Transitions labeled with *sets* of operations and their responses

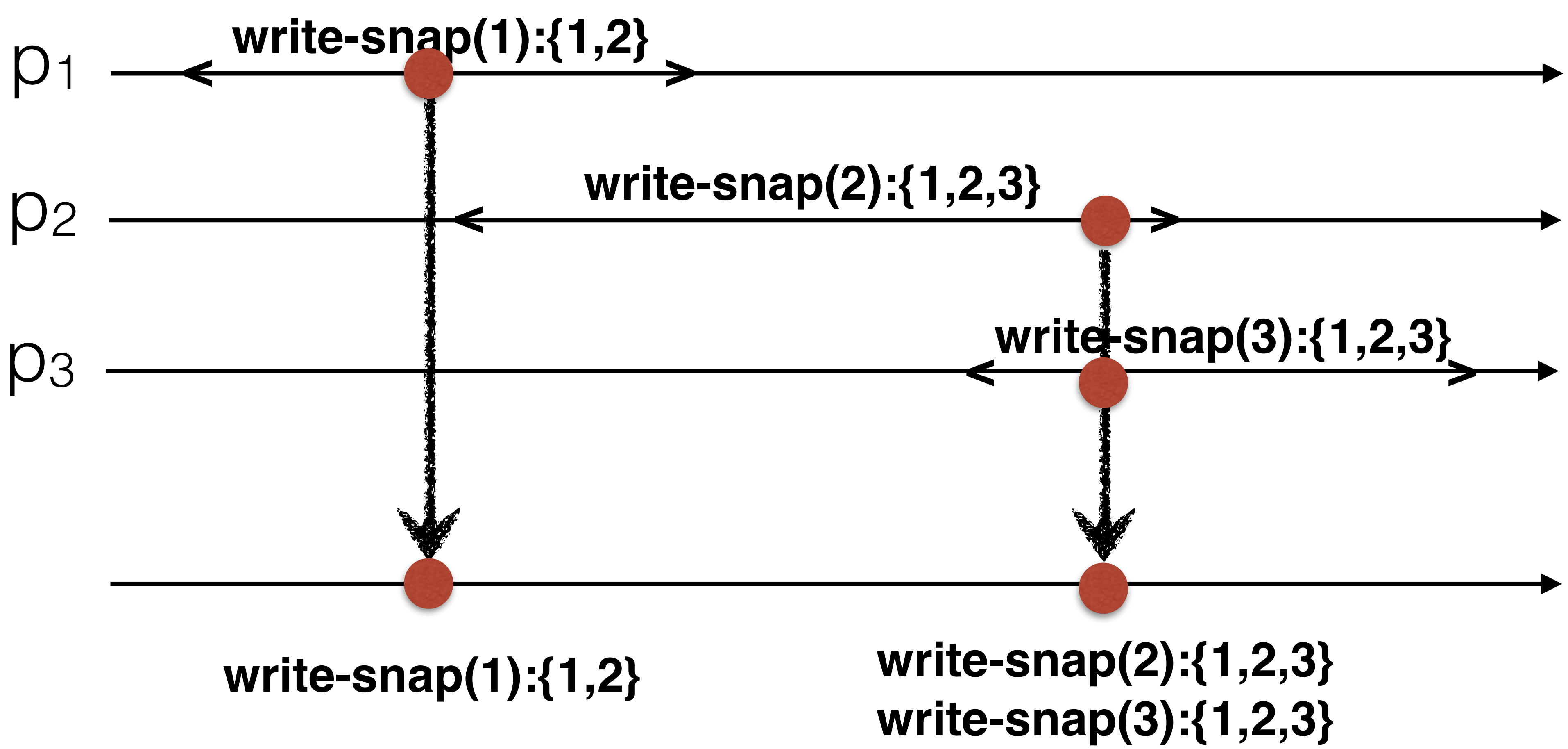


Set linearizability is not enough!!

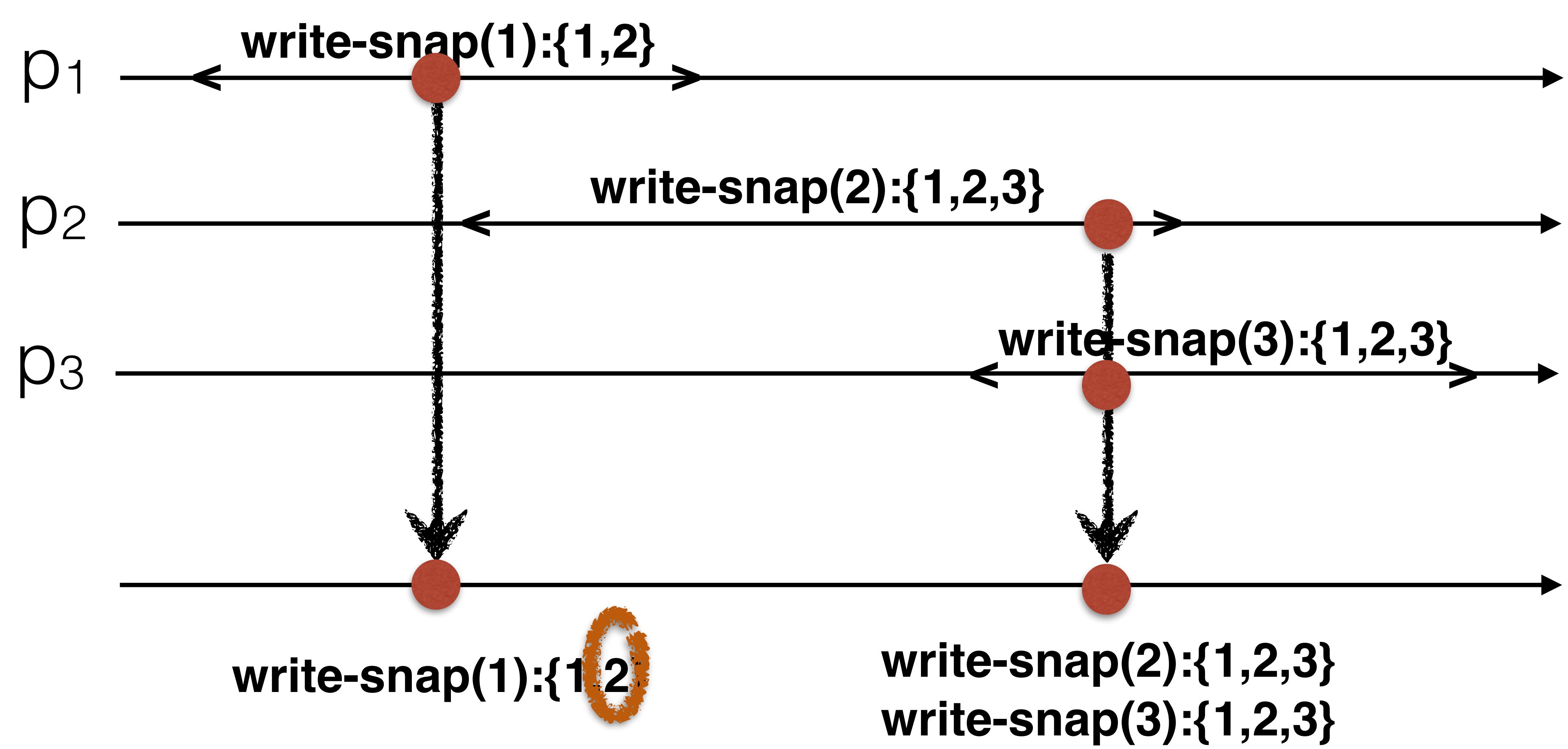
Limitations of Set Linearizability



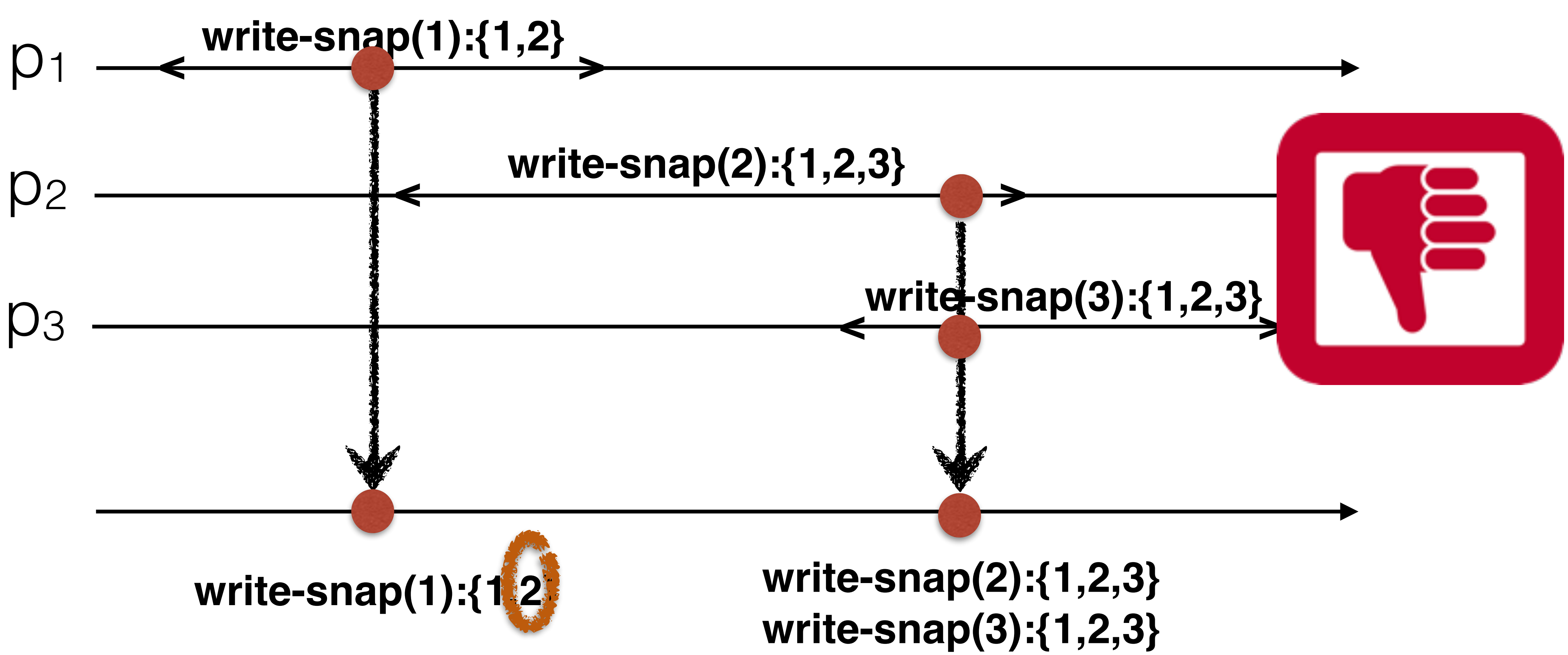
Limitations of Set Linearizability



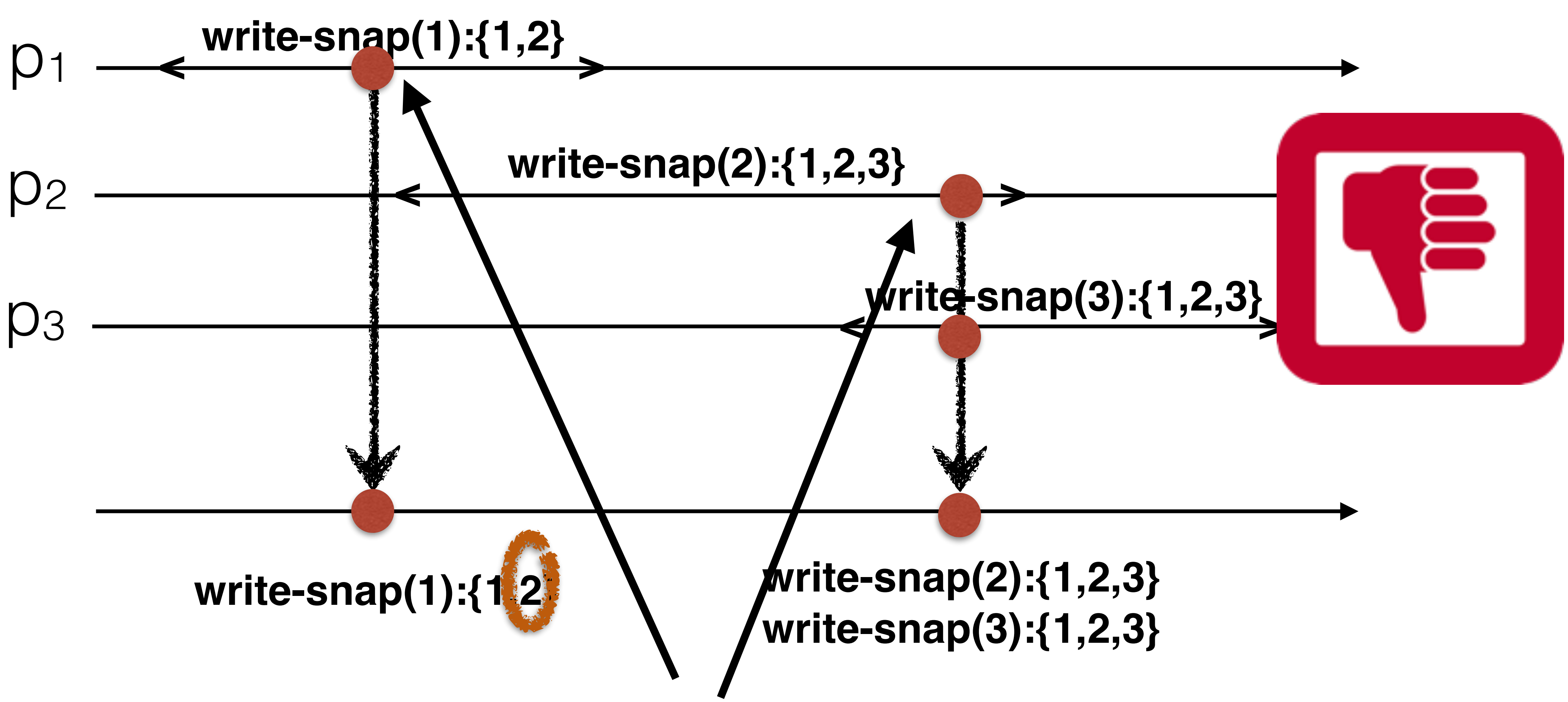
Limitations of Set Linearizability



Limitations of Set Linearizability

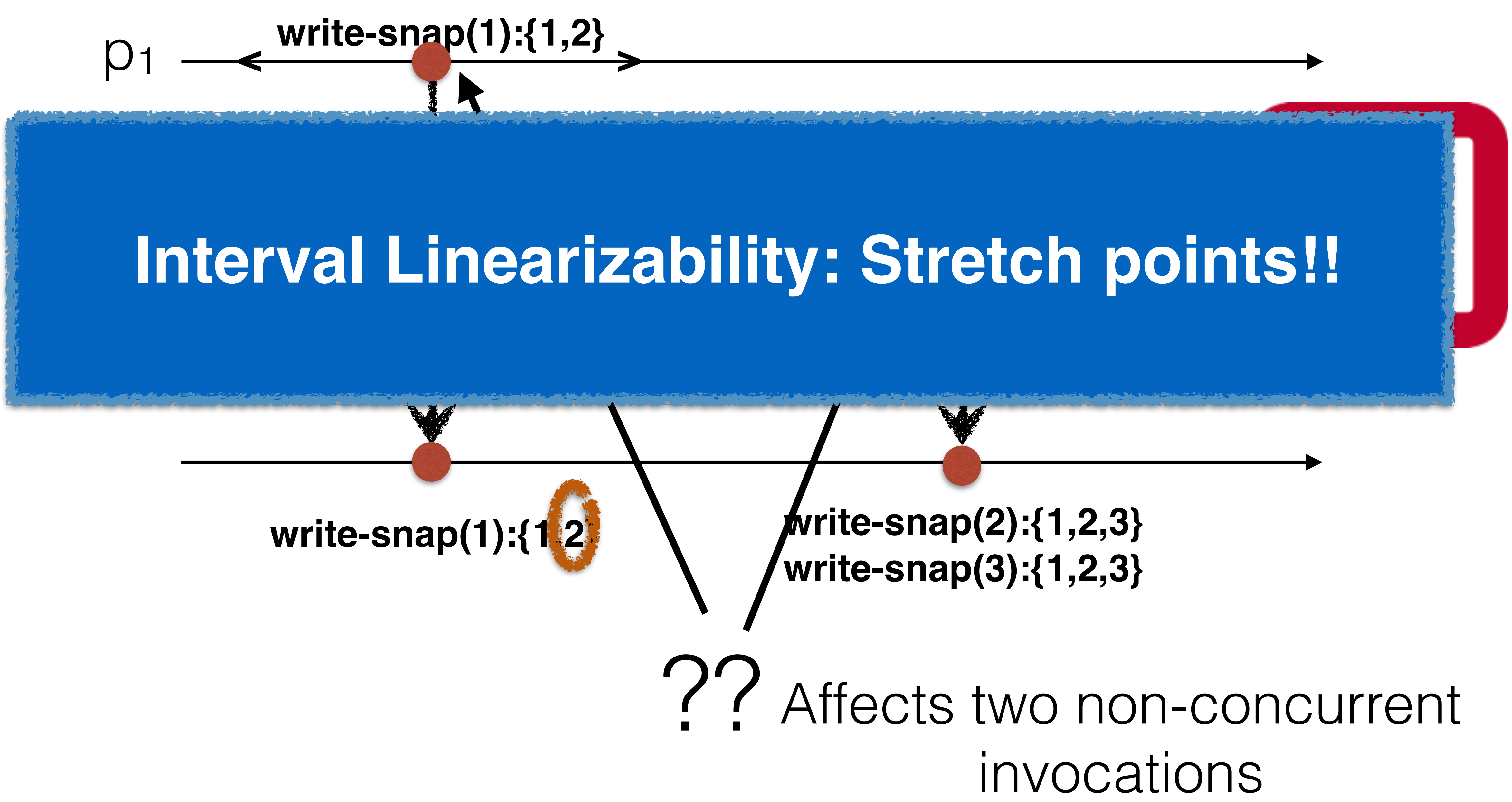


Limitations of Set Linearizability



?? Affects two non-concurrent invocations

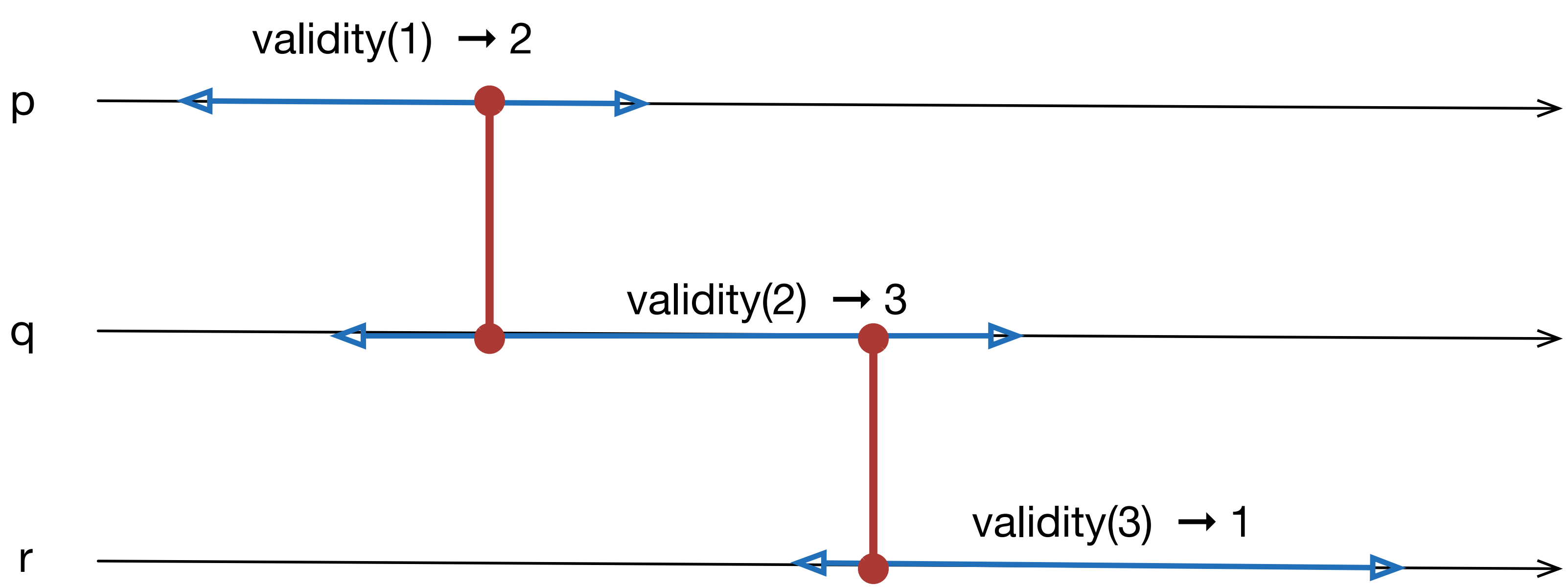
Limitations of Set Linearizability

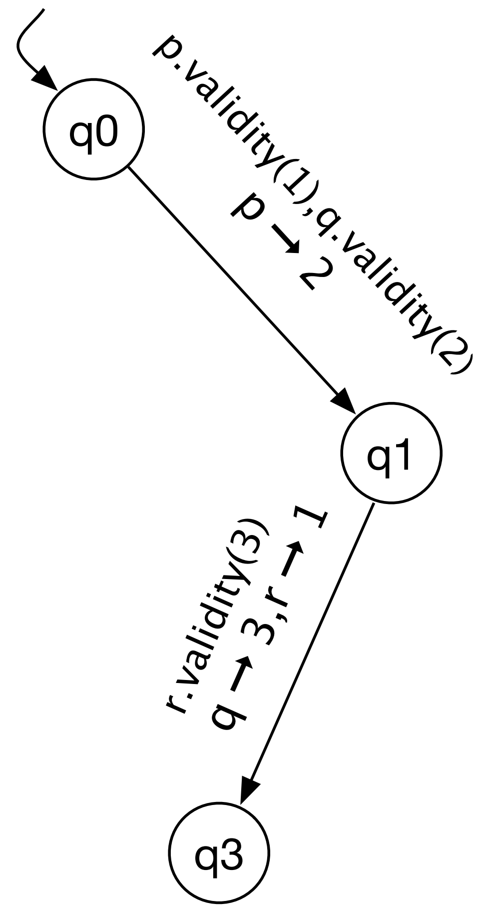


Interval-Sequential automata

- Mealy state machine
- If X is in state q and it receives as input a set of invocations I, then, if $(R,q') \in \delta(q,I)$, the meaning is that X may return the non-empty set of responses R and move to state q'.

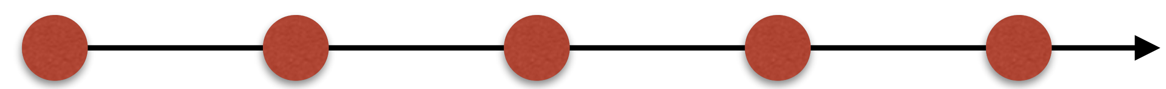
Interval-Sequential Validity Object



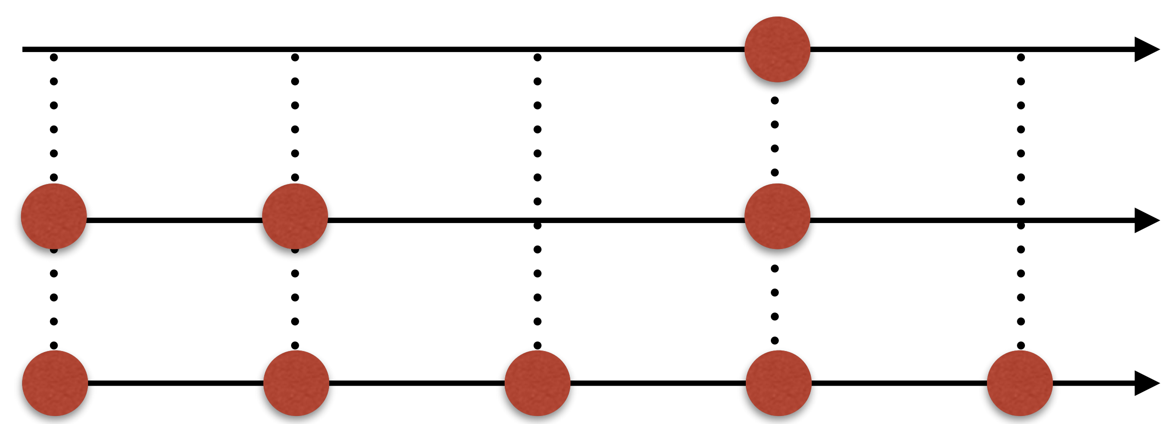


Interval Linearizability

Sequential

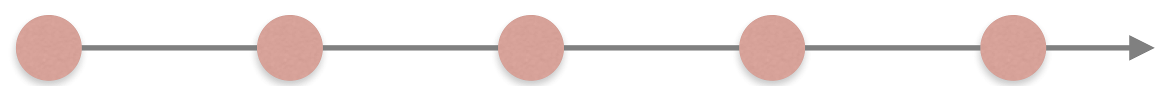


Set sequential

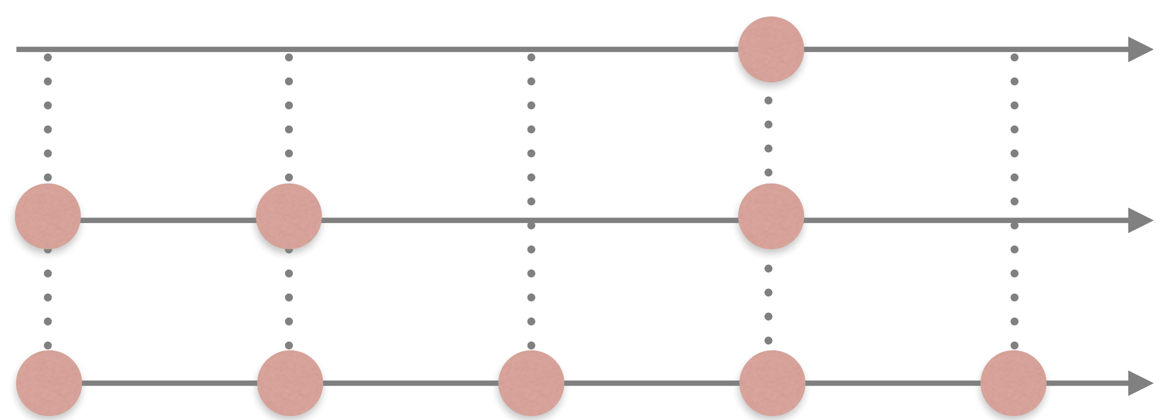


Interval Linearizability

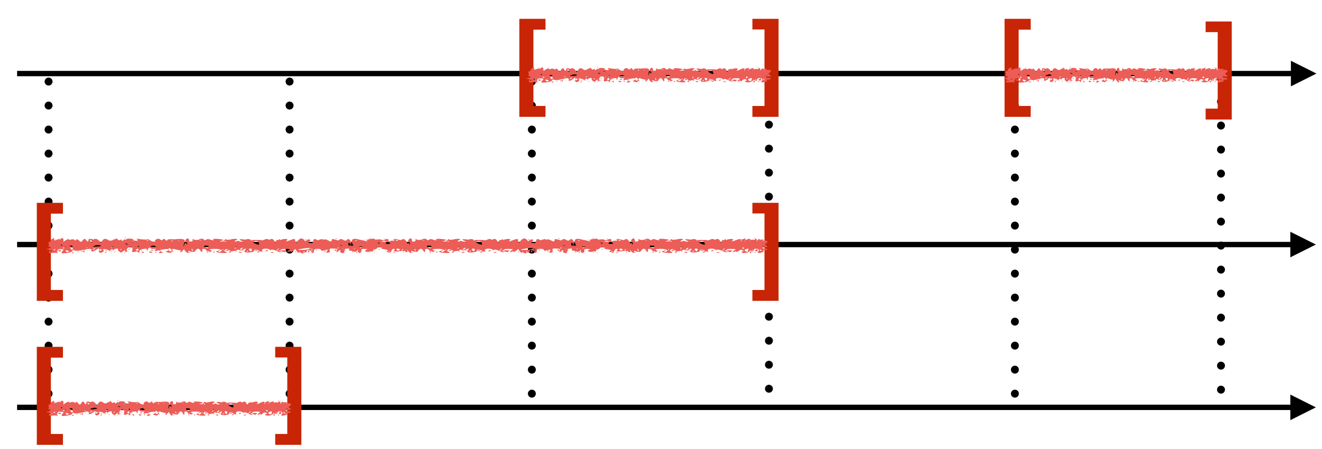
Sequential



Set sequential



Interval sequential



Interval Linearizability

- **Interval Sequential (IS) exec**: Grid with **'nicely'** ordered intervals

  First column: invocations.

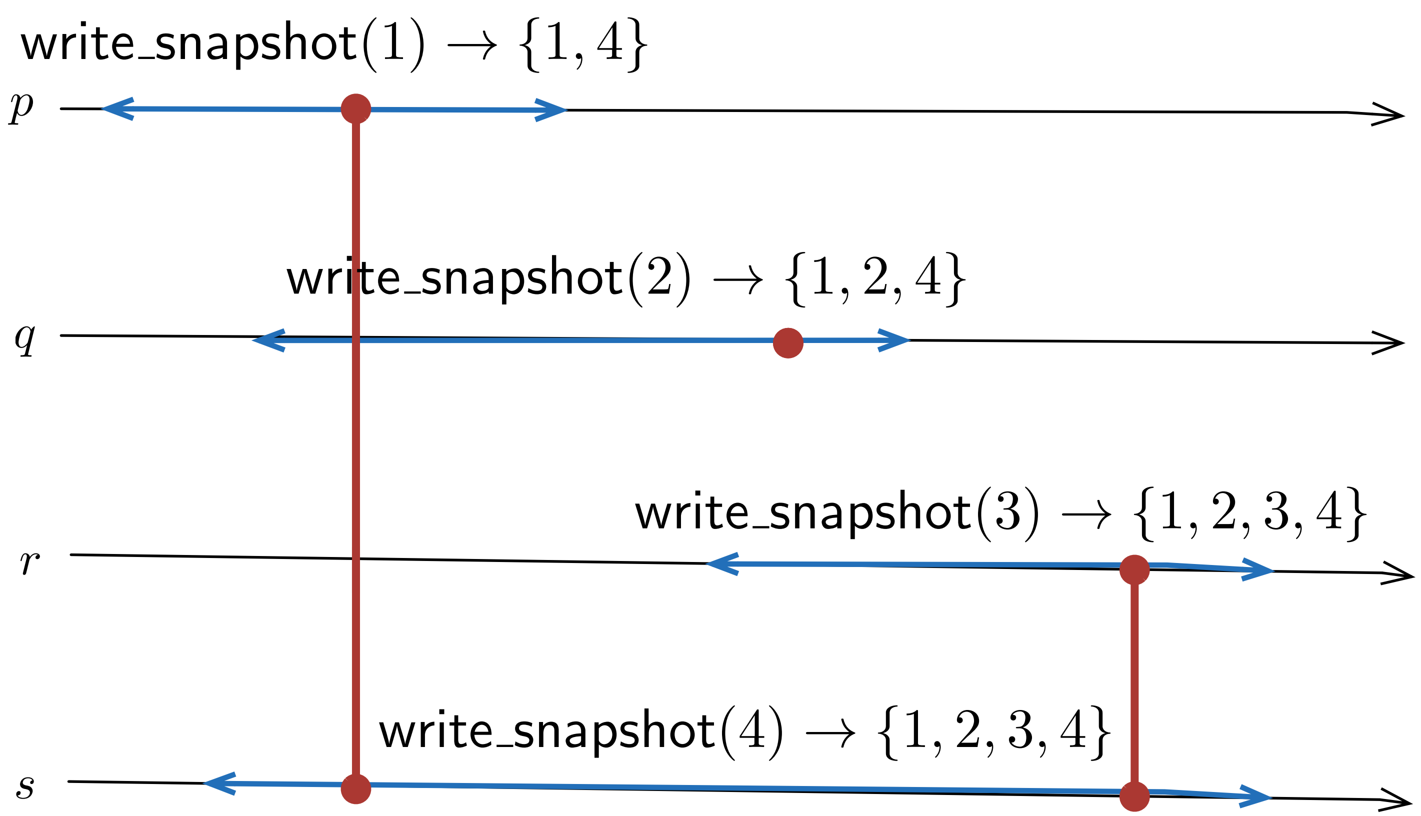
  Second column: responses to some invocations.

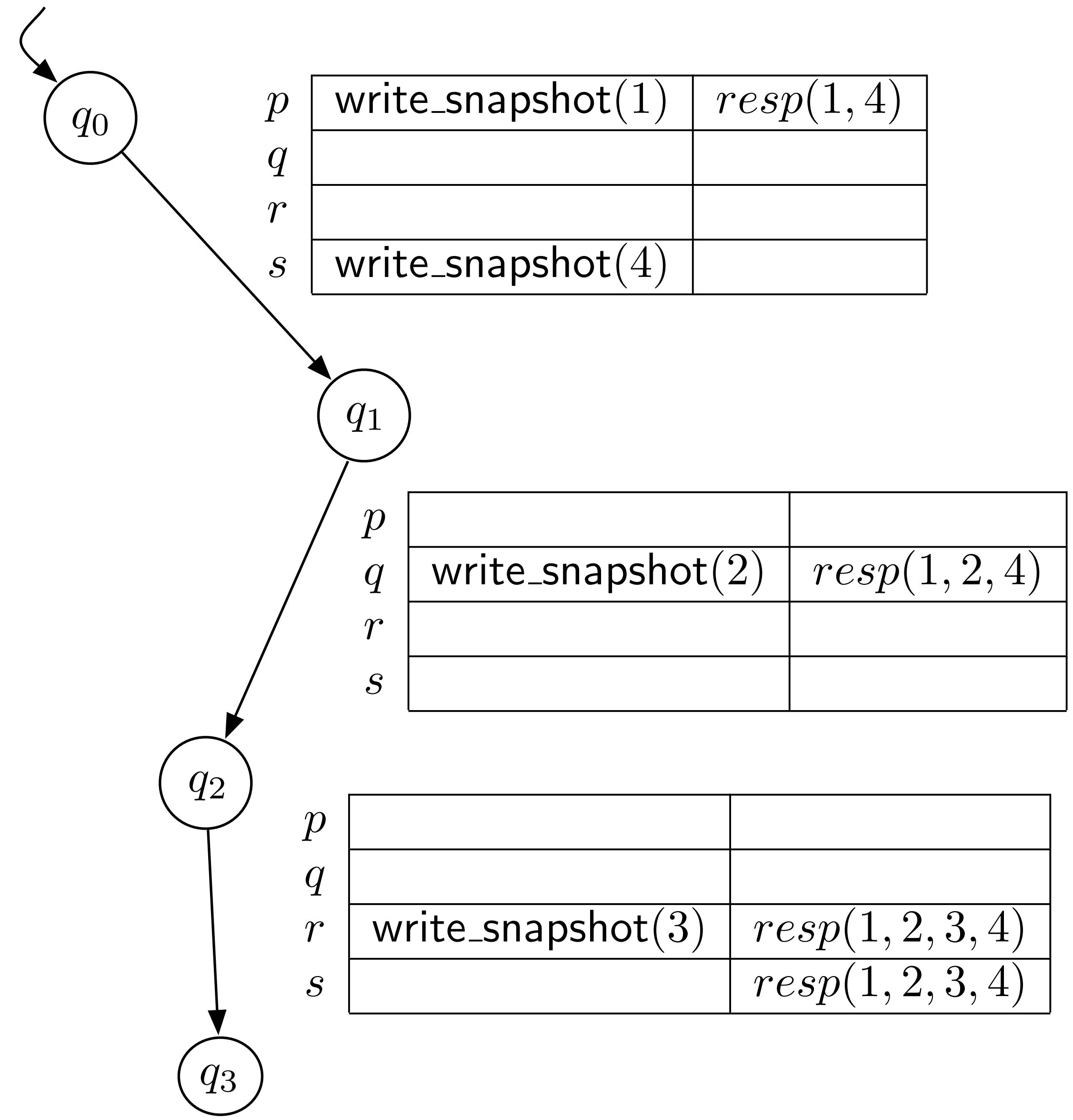
  Third column: new invocations.

  Fourth column: …

- **IS specification**: set with IS executions,
- alternatively IS automaton

Interval Linearizability and automaton





Interval Linearizability

- **Interval linearizable implementation**: each execution can be transformed into a IS execution, respecting real-time order (like in linearizability)

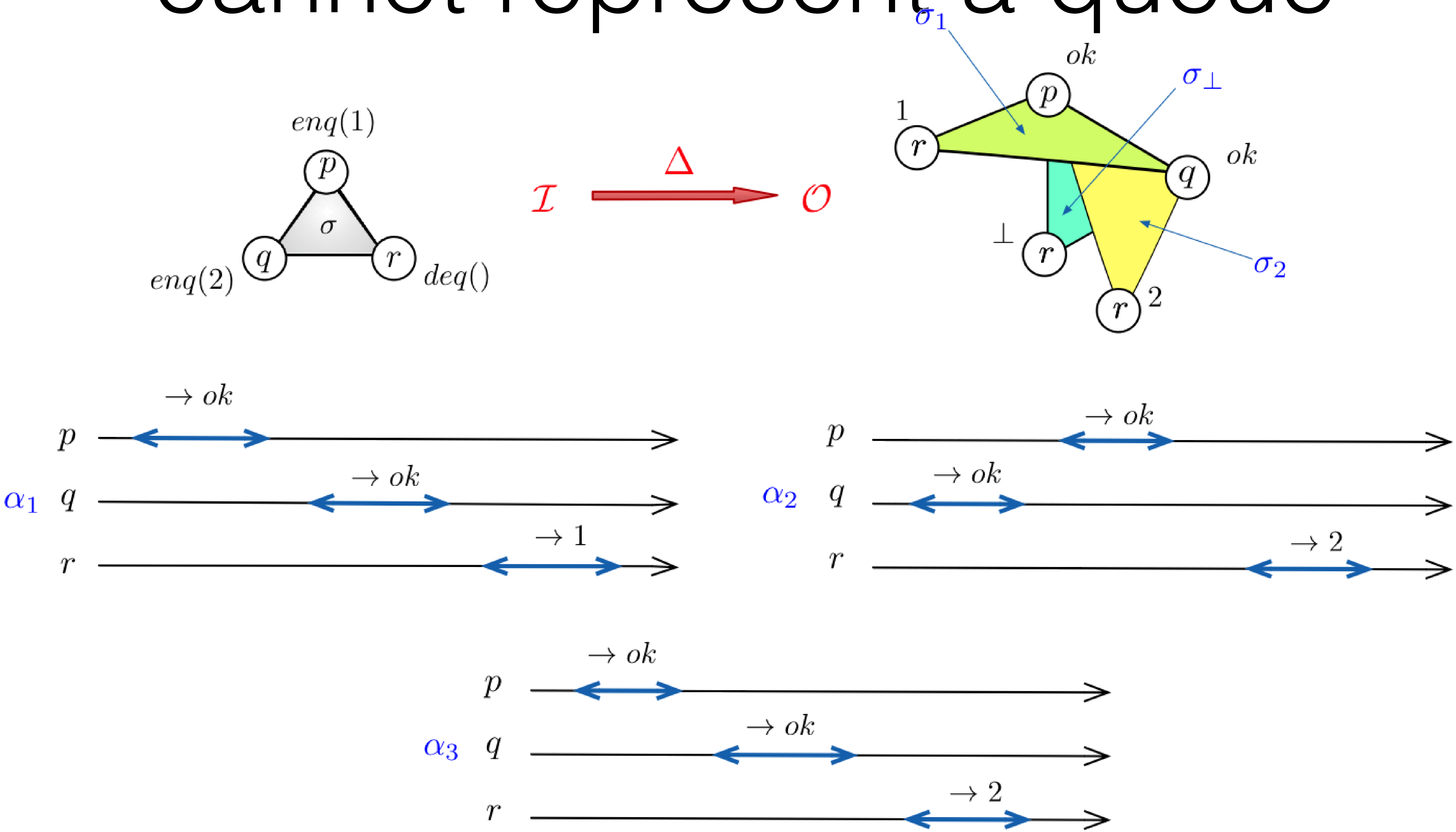
- **Not harder** to prove than linearizability. For each operation, two points (an interval) need to be found

- **Particular cases:** linearizability and set linearizability

Extended Tasks

- A **new value** on each vertex added in the **output complex** to **model memory**
- The **mapping** has the same definition but the **meaning** is **a bit different**
- **Particular case:** Tasks

Simple task interpretation cannot represent a queue



From Interval Linearizability to Extended Tasks

For every one-shot IS object X, there is an extended task equivalent to X

Idea of the proof: Every execution is represented with a simplex of appropriate dimension. New value model memory

By-product: Opens the possibility to apply topological techniques to sequential, set sequential and interval sequential objects.

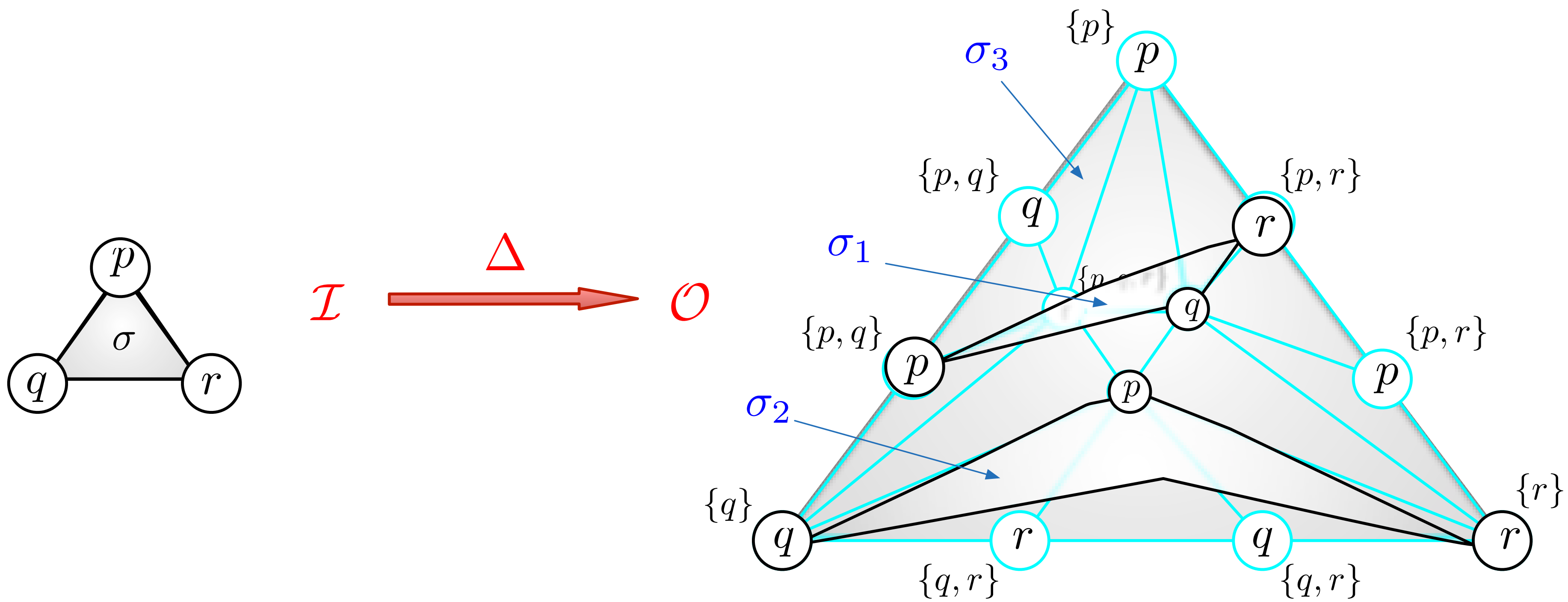
From Extended Tasks to Interval Linearizability

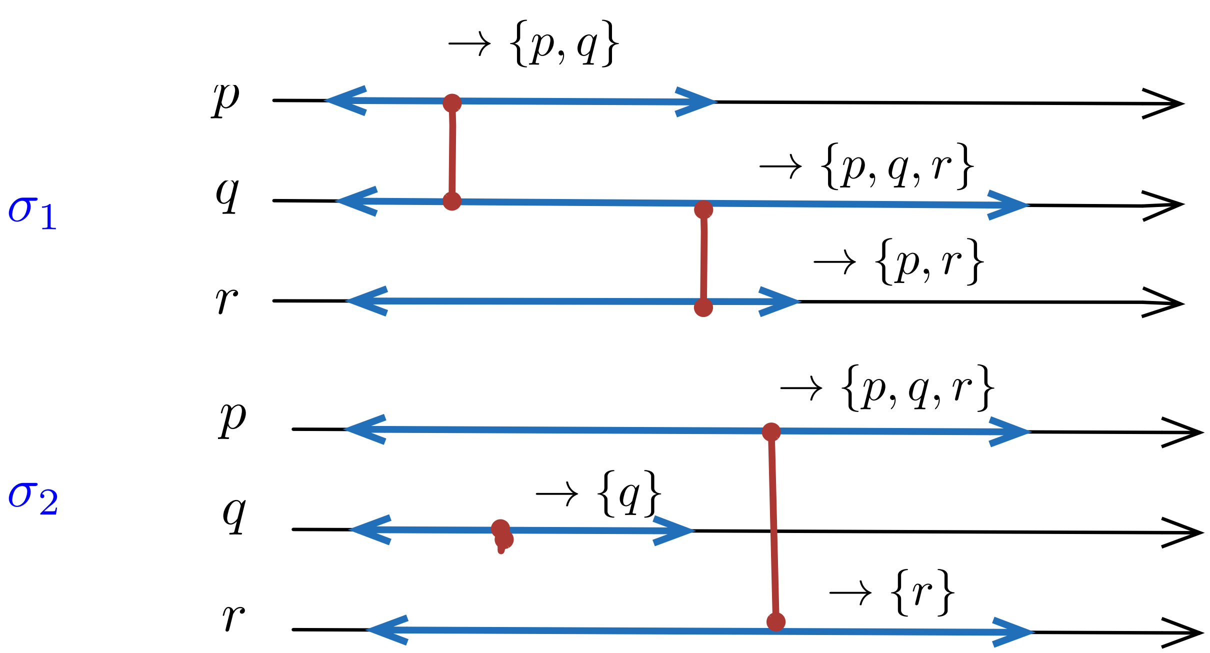
For every extended task T, there is a one-shot IS object equivalent to T

Idea of the proof: Model each output simplex as an IS execution. The interpretation of the mapping from input complex to output complex is not trivial, has to be done carefully.

By-product: Better understanding of the semantics of tasks.

From tasks to interval sequential automata





	init	*term*	*init*	*term*
p	prop(p)	$resp(p,q)$		
q	prop(q)			$resp(p,q,r)$
r			prop(r)	$resp(p,r)$

	init	*term*	*init*	*term*
p			prop(p)	$resp(p,q,r)$
q	prop(q)	$resp(q)$		
r			prop(r)	$resp(r)$

Interval Linearizability Properties

Interval Linearizability Properties

- Local property (like linearizability)

An execution E is interval linearizable if and only if each object X, $E|_X$ is interval linearizable

- Non-blocking property (like linearizability)

For every interval linearizable execution E, there is an interval linearization with all ops in E completed

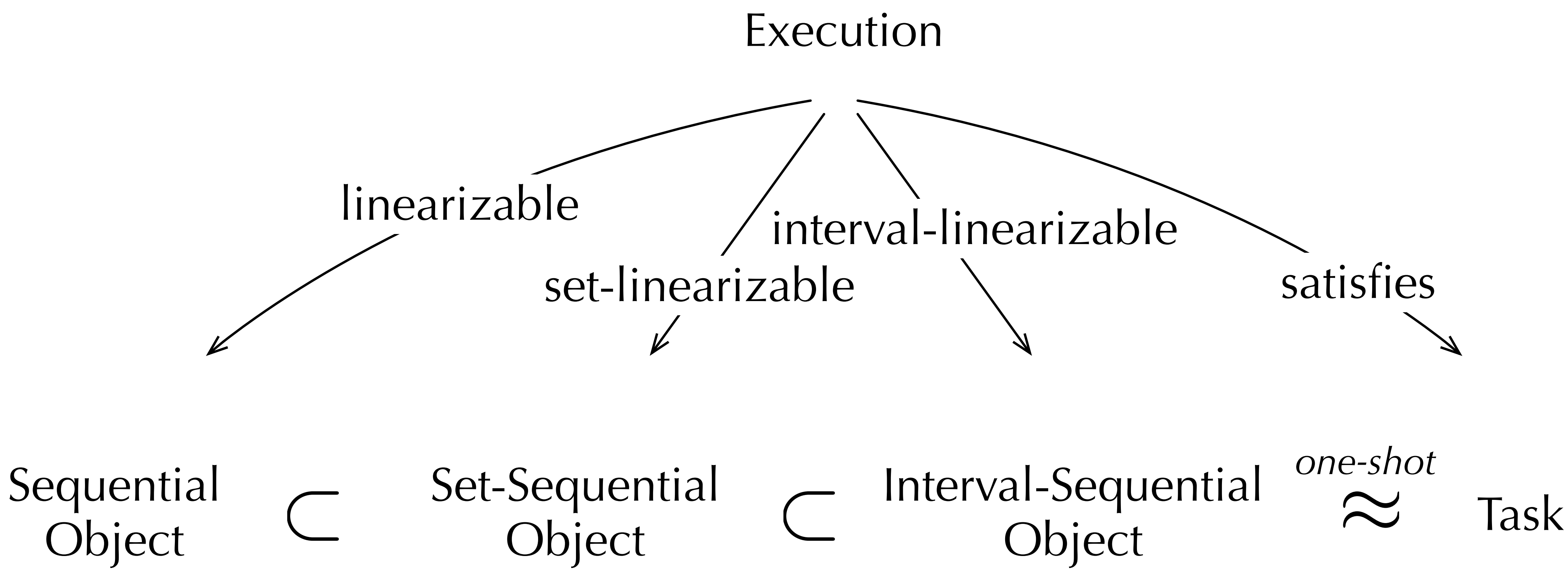
Completness Result

A general definition: Prefix-closed set of executions (with no restrictions, not necessarily one-shot)

Most general definition one can imagine?

For every prefix-closed set of executions, there is a IS object that model the set

Conclusion



- Set-based spec = multi-shot tasks = IS linearizability
- We are working on extend task definition further, to model multi-shot objects
- and on applying topological techniques to objects

Thanks!!