Immediate Snapshot

CA 2017

Jingjing Wang, LPD, EPFL

Snapshot

A **snapshot** has two operations: ***update()*** and ***scan()*** and maintains an array x of size n

Sequential specification

scan():

- Return (x)

update(i, v):

- x[i] := v;
- Return (OK)

Motivation for immediate snapshot

Snapshot

- Update some state
- Take a “picture” of all states
- Separately

Immediate snapshot

- Immediately take a “picture” of all states after updating a state

Semantics

The memory is accessed via a single **update_snapshot** operation

Semantics: each write operation, in addition to writing, also returns an atomic snapshot

“Weakly atomic” = runs of standard atomic snapshot include runs of immediate snapshot

The power of registers

Can immediate snapshot be implemented by atomic registers?

- Yes. At least for one-shot version

One-shot: Each process invokes at most once that operation

Immediate snapshot

An **immediate snapshot** has a single operation: ***update_snapshot()*** and maintains an array x of size n

Sequential specification

update_snapshot(vi):

- x[i] := vi;
- Return {(1, x[1]), (2, x[2]), …, (n, x[n])}

Properties

Liveness. An invocation of **update_snapshot()** terminates

Self-inclusion. $(i, v_i) \in view_i$

Containment. $view_i \subseteq view_j$ or $view_j \subseteq view_i$

Immediacy. If $(j, v_j) \in view_i$, then $view_j \subseteq view_i$

Naive implementation

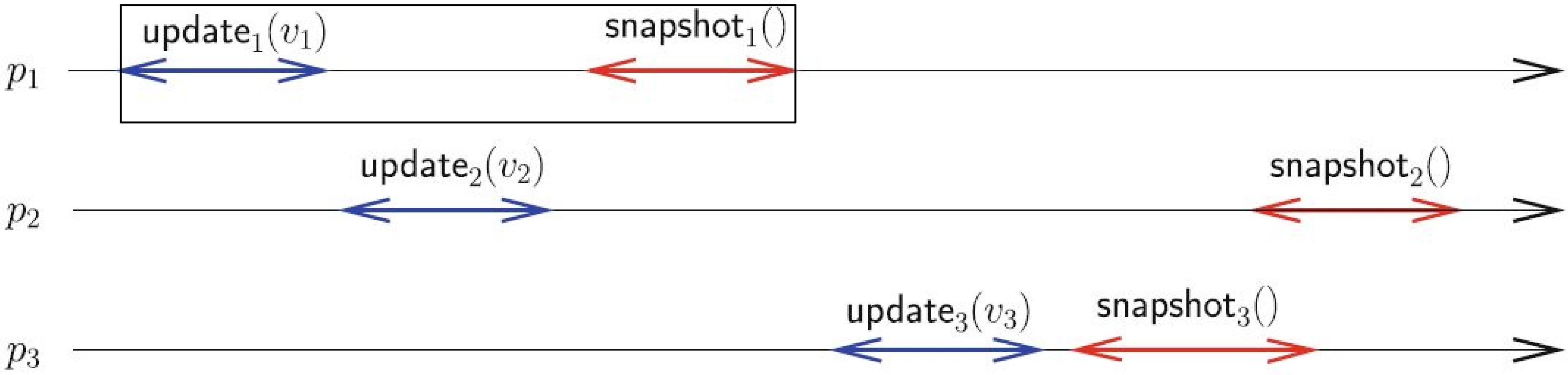
n processes share an atomic snapshot object x

update_snapshot(vi):

- x.update(i, vi);
- a := x.scan();
- Return {(1, a[1]), (2, a[2]), …, (n, a[n])}

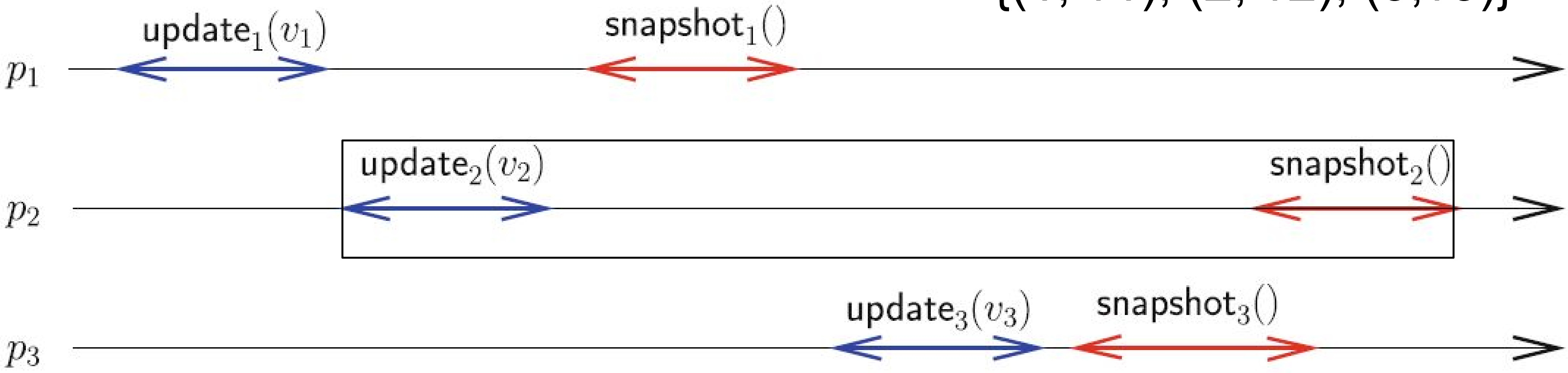
Immediacy?

update_scan() - {(1, v1), (2, v2)}



Immediacy?

update_scan()
- {(1, v1), (2, v2), (3,v3)}



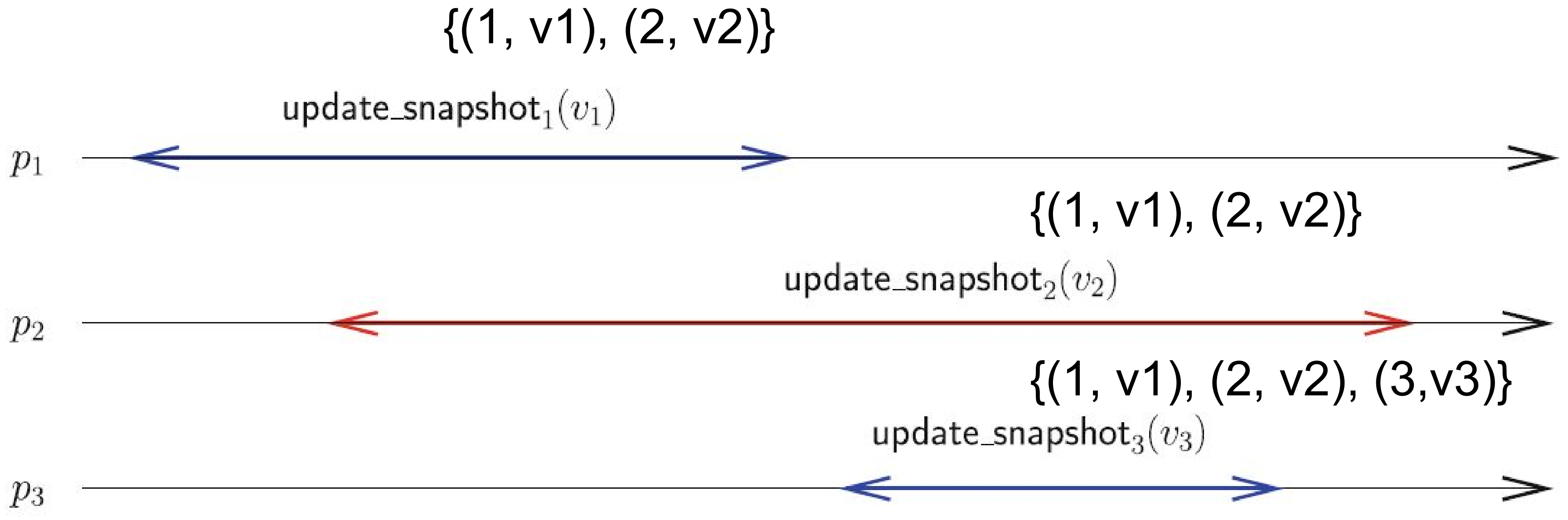
Snapshot vs. immediate snapshot

An atomic snapshot

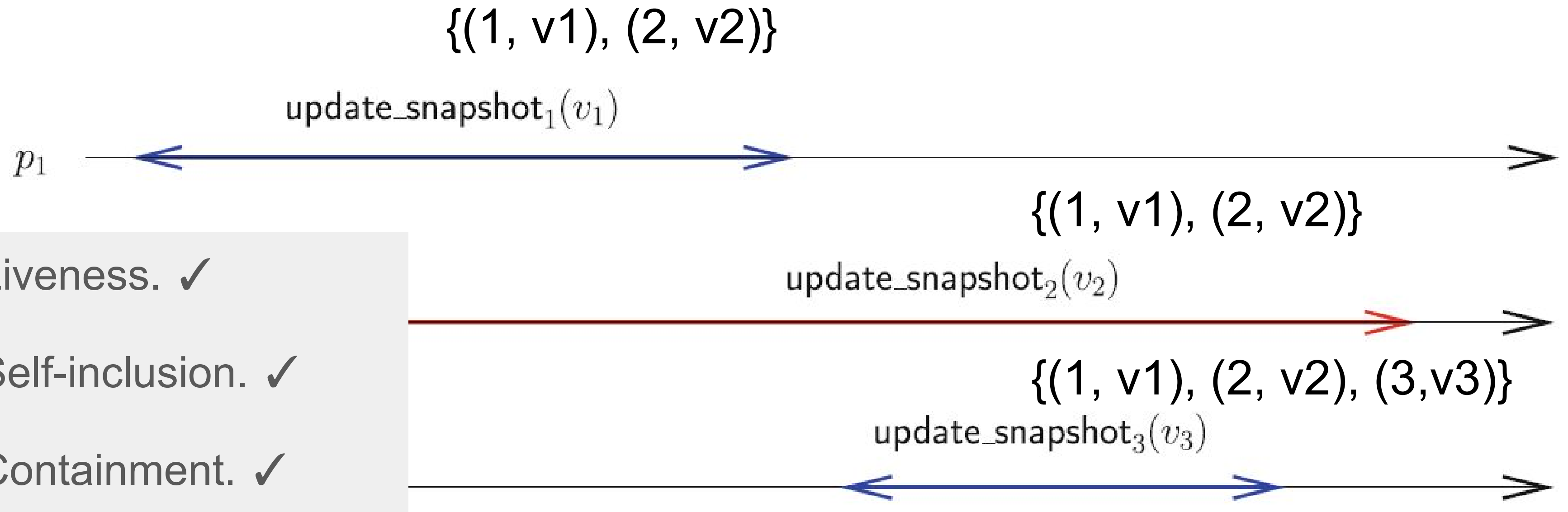
An immediate snapshot that satisfies

- Liveness, self-inclusion, containment, immediacy

Possible execution?



Possible execution?



Liveness. ✓

Self-inclusion. ✓

Containment. ✓

Immediacy. ✓

A property that follows

(Self-inclusion. (i, vi) ∈ viewi

+ Immediacy. If (j, vj) ∈ viewi, then viewj ⊆ viewi)

Property: If (i, -) ∈ viewj and (j, -) ∈ viewi, then viewj = viewi

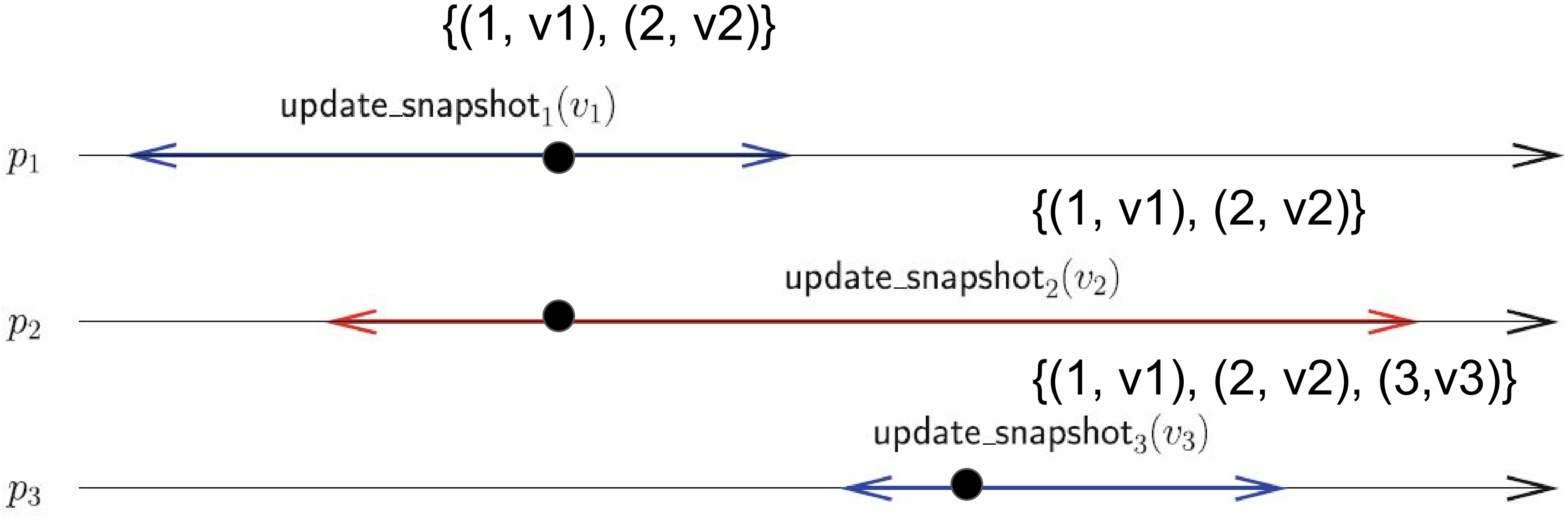
=> Compared with sequential execution?

Atomicity

Every operation appears to execute at

- Some indivisible point in time (called linearization point) between
- The invocation and reply time events

Atomic execution?



Set linearizability

Linearization replaced by set-linearization:

- These invocations are set-linearized at the same point of the time line

For one-shot immediate snapshot,

- The invocations which are set-linearized at the same point do return the very same view

Key idea for set linearizability

To ***update_scan()***, a process keeps reading other processes' updates

For any two processes pi and pj,

- If pi and pj see each other's update, then pi and pj retry reading until they are going to return the **same** result

Enforcing set linearizability

The processes share an array of ***registers*** REG[1], REG[2], REG[3], …

- REG[x] is again an array of registers
- REG[x] contains a view
- REG[x][i] can only be written by pi

Pi reads REG[x]

- If pi cannot return REG[x], then pi retries, writes and reads the **next** REG

Enforcing set linearizability

The processes share an array of ***registers*** REG[1], REG[2], …, init'ed to ⊥

A recursive implementation:

- ***update_snapshot(vi):***
    - my_viewi := rec_update_snapshot(**next**, vi)
    - Return my_viewi

Enforcing set linearizability

Every process keep a local array of registers Regi

- ***rec_update_snapshot(x, v):***
    - REG[x][i].write(v);
    - For each j ∈ {1,..., n} do Regi[j] := REG[x][j].read();
    - Viewi := { (j, Regi[j]) | Regi[j] ≠ ⊥};
    - if(**some condition**) then resi := viewi;
    - Else resi := rec_update_snapshot(**next**, v);
    - Return resi

Possible execution?

	REG[1]	REG[2]	REG[3]	...	
p1	v1	v1	v1		
p2	v2	v2	v2		
p3		v3	v3		

Key idea for liveness

If pi and pj see each other’s update, then pi and pj retry

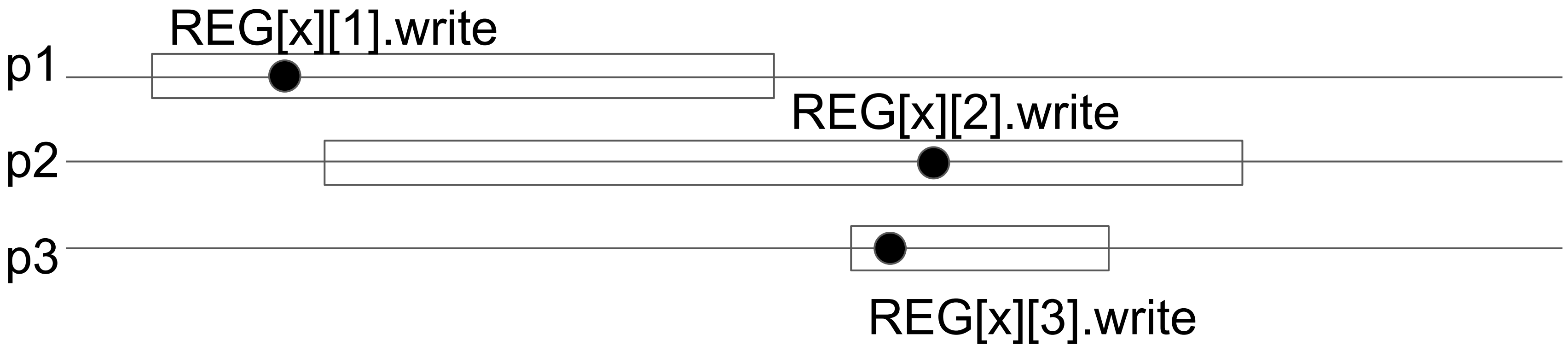
- Pi is waiting for pj’s last-minute view
- So is pj
- Which view is the last one?

Key idea for liveness (cont'd)

Suppose: At most x processes access REG[x] (invariant)

If pi sees REG[x] contains exactly x updates, then

- pi is one of the **last** processes which accesses REG[x]
- Or linearized as such



Key idea for liveness (cont'd)

Suppose: At most x processes access REG[x] (invariant)

If pi sees REG[x] contains exactly x updates, then

- pi is one of the **last** processes which accesses REG[x]
- Or linearized as such

If the invariant is true, then after pi, REG[x] remains the **same**.

Key idea for liveness (cont’d)

Suppose: At most x processes access REG[x] (invariant)

If pi sees REG[x] contains exactly x updates, then

- pi is one of the **last** processes which accesses REG[x]
- Or linearized as such

If the invariant is true, then after pi, REG[x] remains the **same**

- Pi can return REG[x]
- As well as other processes who see pi’s update

Key idea for set-linearizability & liveness

Recall that we consider one-shot version:

- Each process invokes at most once ***update_scan()***
- This means at most n processes access the **first** REG

Key idea for set-linearizability & liveness

Recall that we consider one-shot version:

- Each process invokes at most once ***update_scan()***
- This means at most n processes access the **first** REG = REG[n]

If **some condition** = a process's view of REG[n] contains n values, then

- Return REG[n]
- Otherwise, go to the **next** REG = REG[n-1]

Key idea for set-linearizability & liveness (cont'd)

The processes share an array of ***registers*** REG[n], REG[n-1], …, REG[1]

- Each contains a view

Claim:

(a) At most x processes can access REG[x]
(b) At least one process return REG[x]

Immediate snapshot implementation

- ***update_snapshot(vi):***
  - my_viewi := rec_update_snapshot(n, vi)
  - Return my_viewi

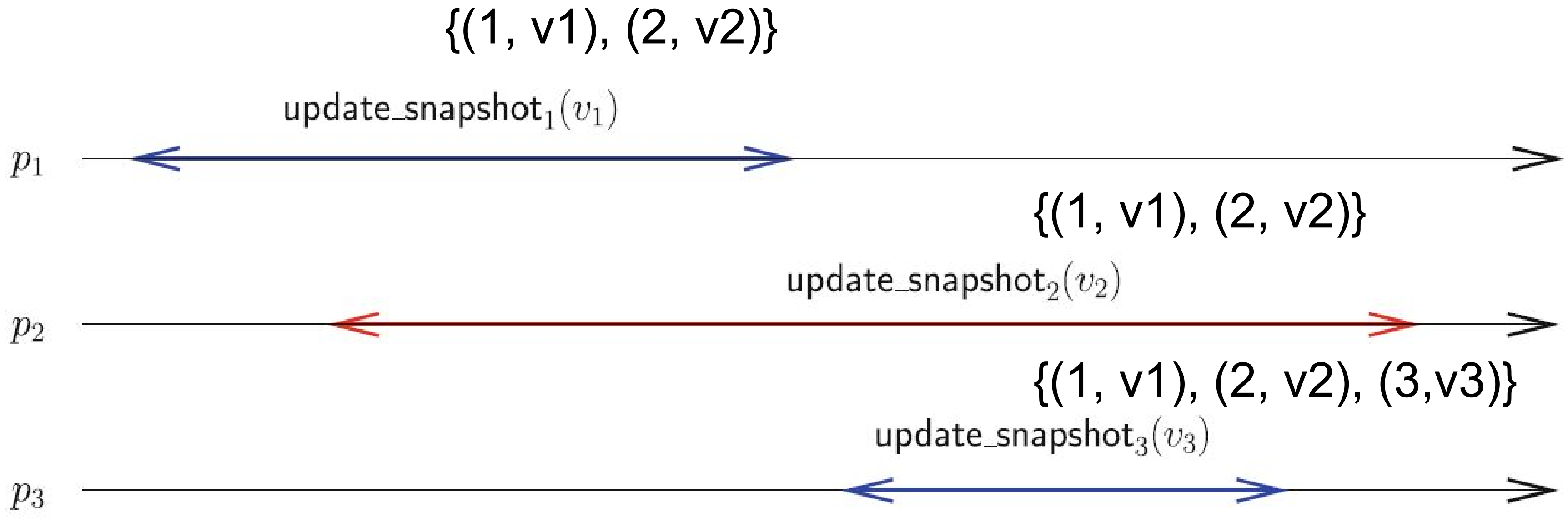
Immediate snapshot implementation

The processes share an array of ***registers*** REG[1, …, n], init'ed to ⊥

Every process keeps a local array of registers Regi

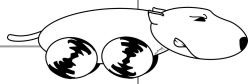
- ***rec_update_snapshot(x, v):***
  - REG[x][i].write(v);
  - For each j ∈ {1,..., n} do Regi[j] := REG[x][j].read();
  - Viewi := { (j, Regi[j]) | Regi[j] ≠ ⊥};
  - if(|viewi| = x) then resi := viewi;
  - Else resi := rec_update_snapshot(x-1, v);
  - Return resi

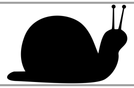
Possible return value?



Possible execution?

	REG[n]	REG[n-1]	...	REG[3]	REG[2]
p1	v1	v1	...	v1	v1
p2	v2	v2	...	v2	v2
p3	v3	v3	...	v3	





References

[1] Elizabeth Borowsky and Eli Gafni. 1993. Immediate atomic snapshots and fast renaming. In Proceedings of the twelfth annual ACM symposium on Principles of distributed computing (PODC '93). ACM, New York, NY, USA, 41-51. DOI=http://dx.doi.org/10.1145/164051.164056

[2] Raynal M. (2013) Snapshot Objects from Read/Write Registers Only. In: Concurrent Programming: Algorithms, Principles, and Foundations. Springer, Berlin, Heidelberg