Implementing Snapshot Objects on Top of Crash-Prone Asynchronous Message-Passing Systems

Carole Delporte-Gallet, Hugues Fauconnier, Sergio Rajsbaum, Michel Raynal

- Processes communicate by applying operations on and receiving response from shared objects

- A shared objects is define by:
  - states
  - operations
  - sequential specification

Register

- Operations: read, write
- state: item
- sequential specification:

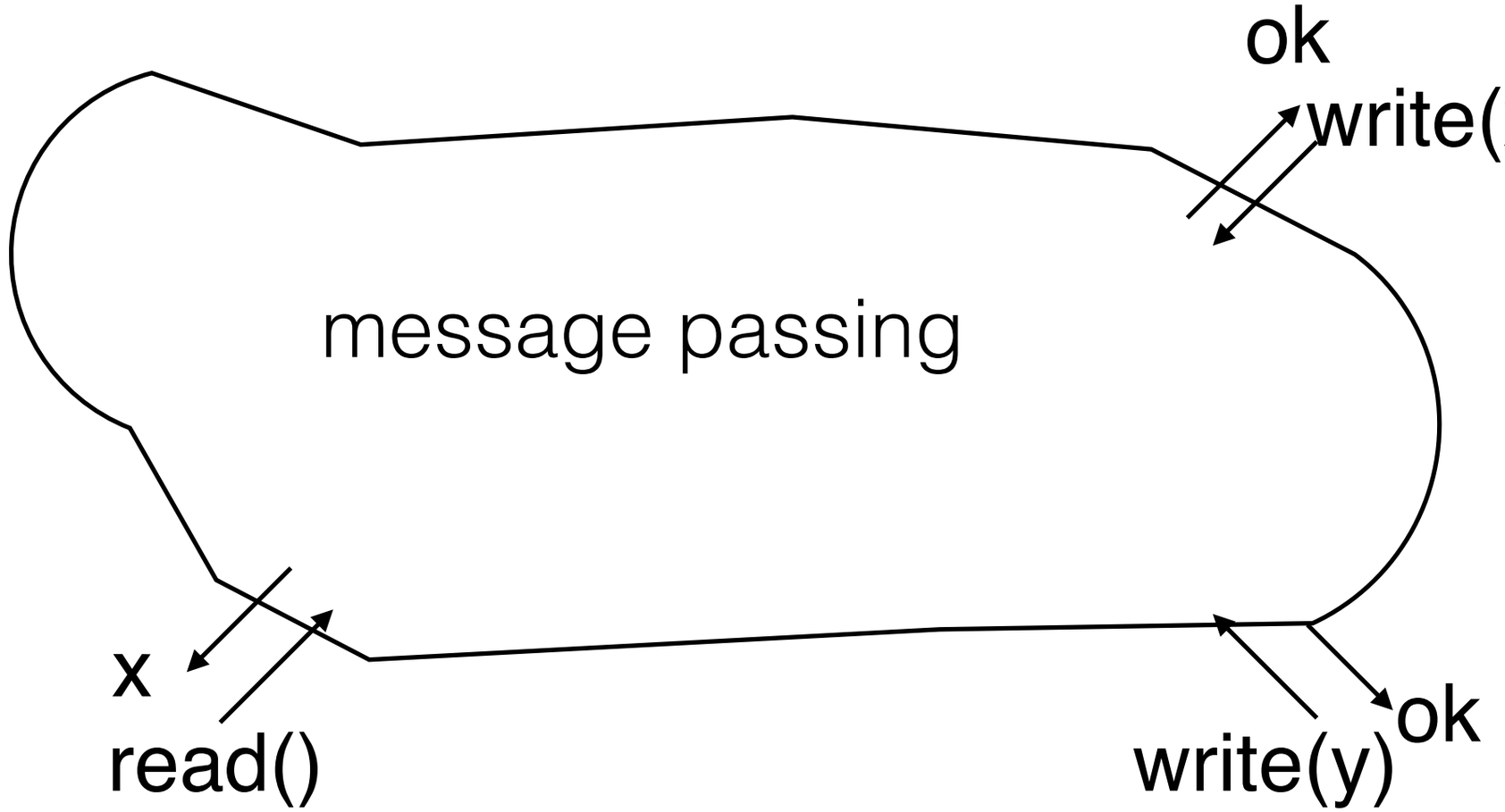
$$\{state = y\} write(x) \{state = x\} \qquad return\ ok$$

$$\{state = y\} read() \{state = y\} \qquad return\ y$$

Implementation of object O

- An operation on O is implemented using messages
- Correctness ?

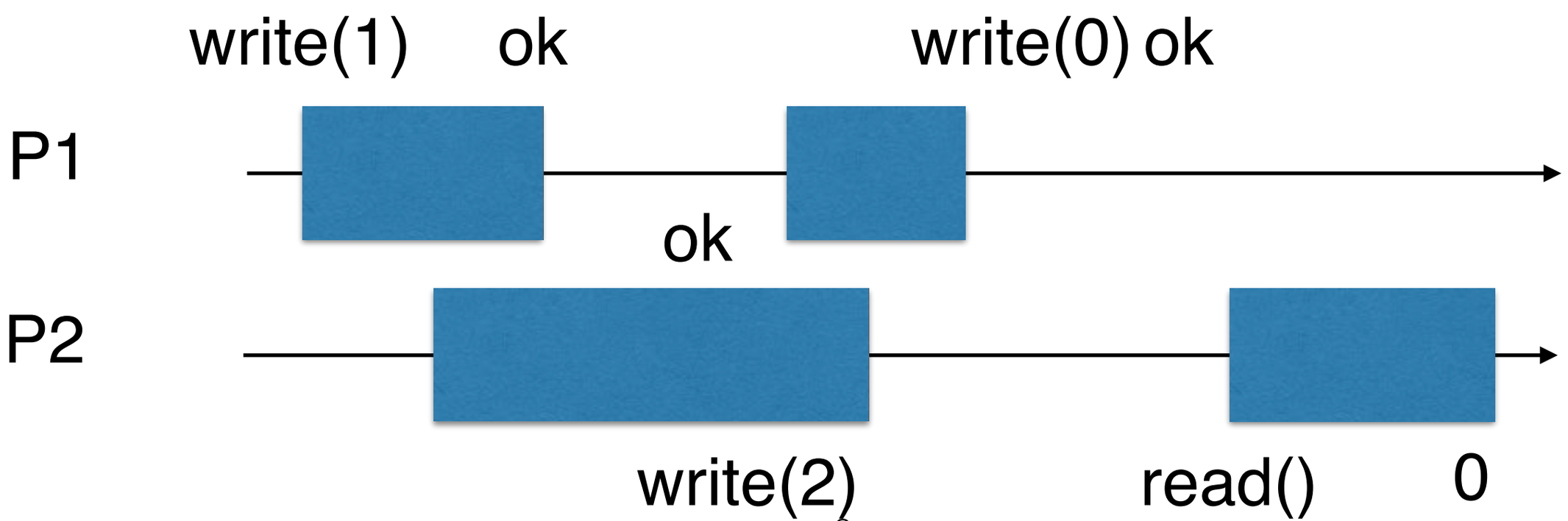
Implementing an object

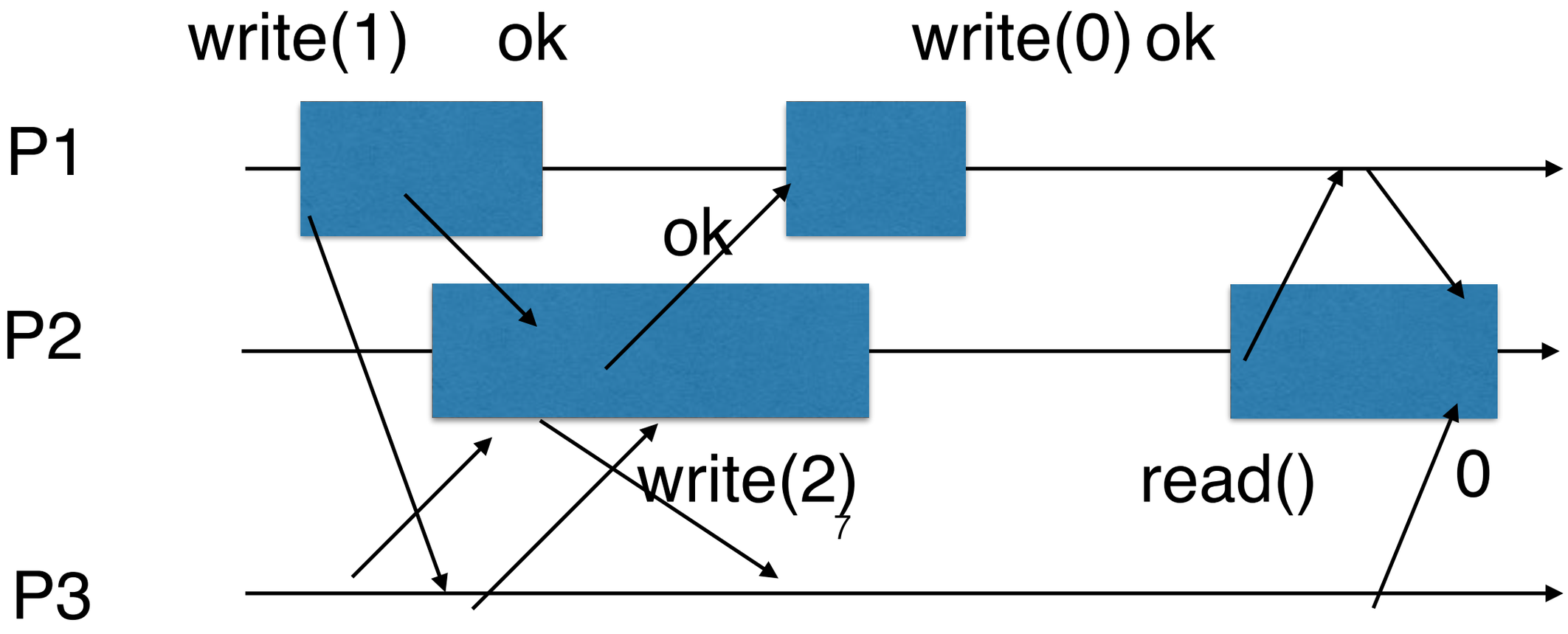


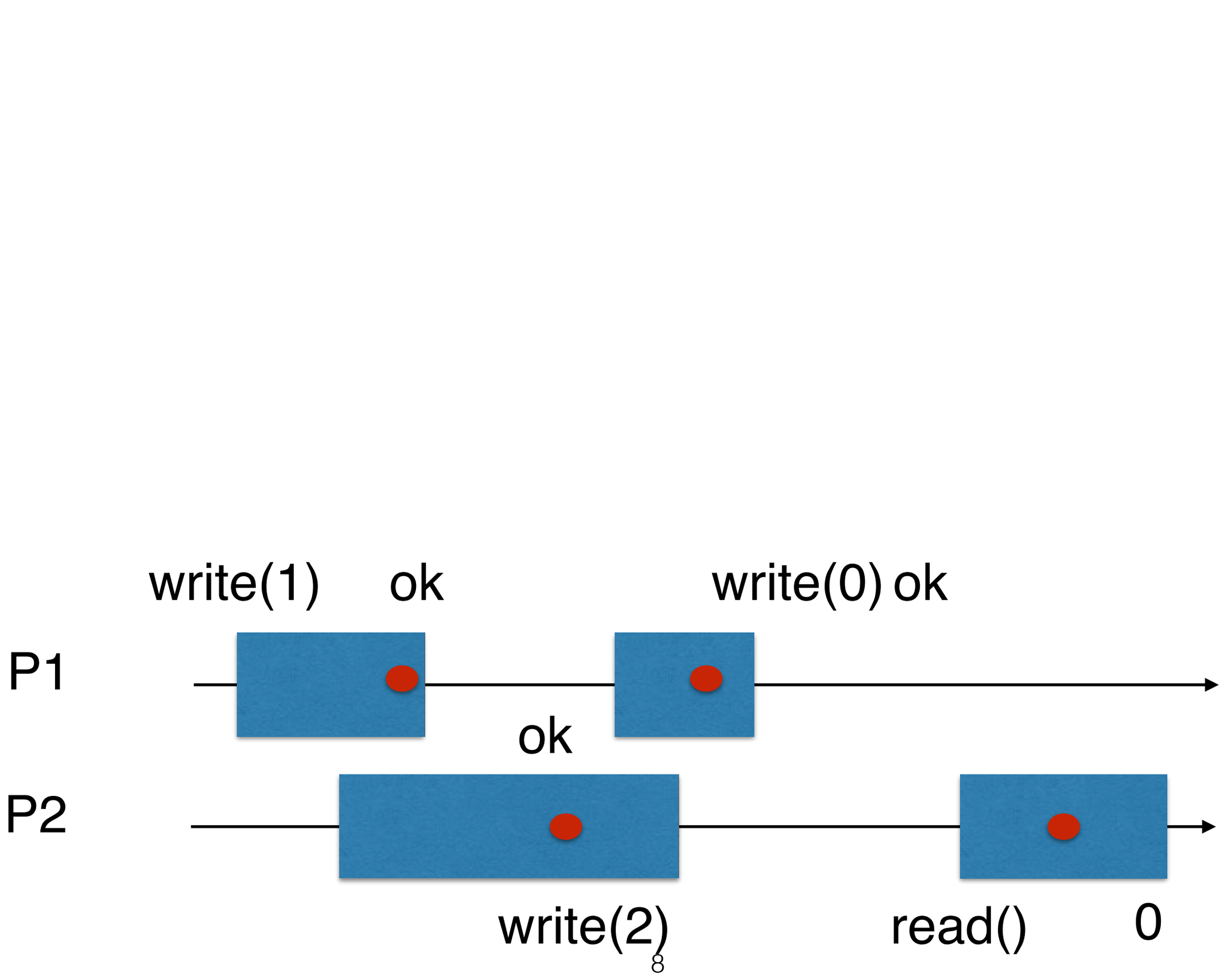
Histories

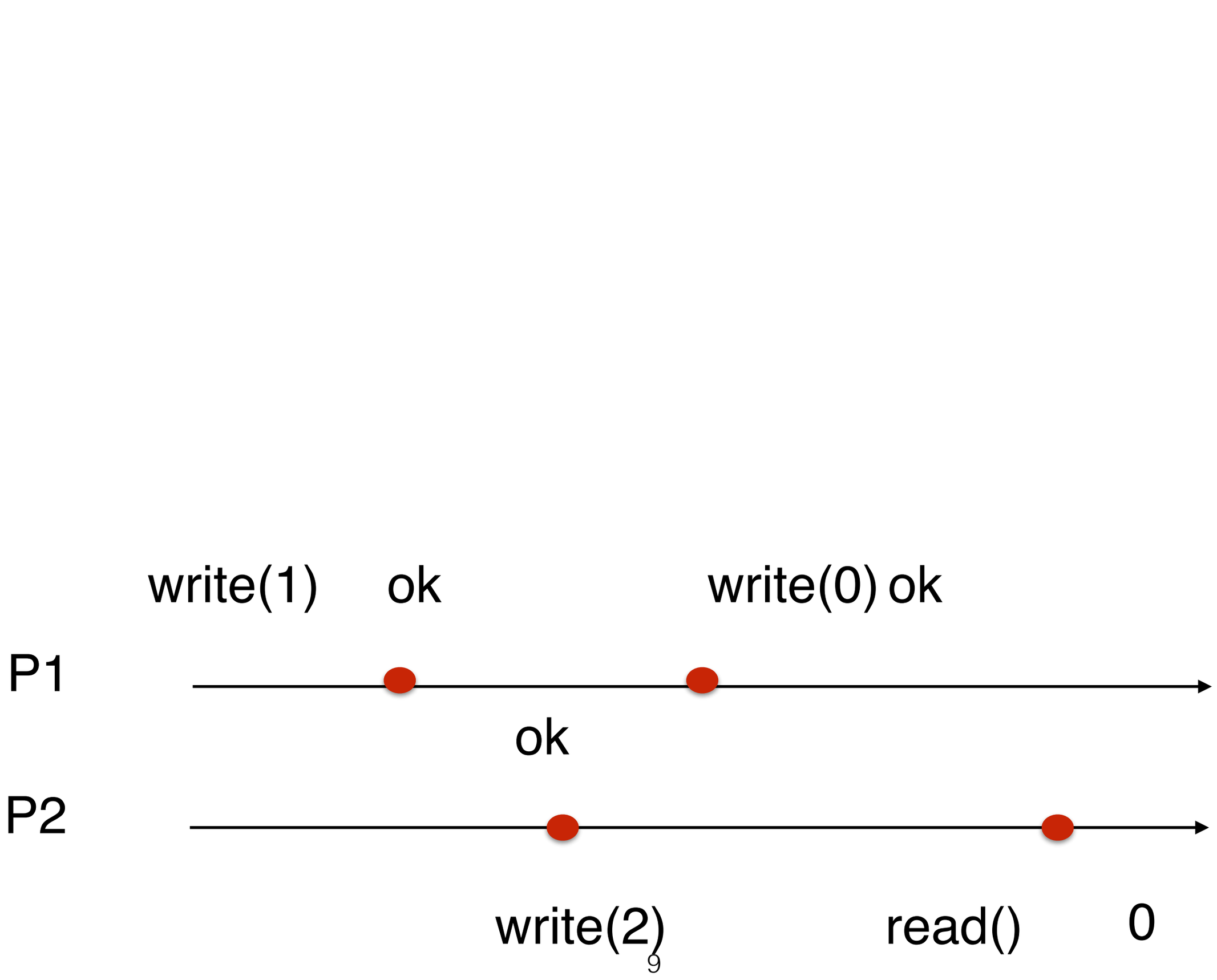
- A history is a sequence of invocation and response

$$write_1(1) write_2(2) ok_1 write_1(0) ok_2 ok_1 read_2() 0_2$$




write(1)
ok
write(0) ok
P1
ok
P2
write(2)
read()
0
P3


write(1)
ok
write(0) ok
P1
ok
P2
write(2)
read()
0

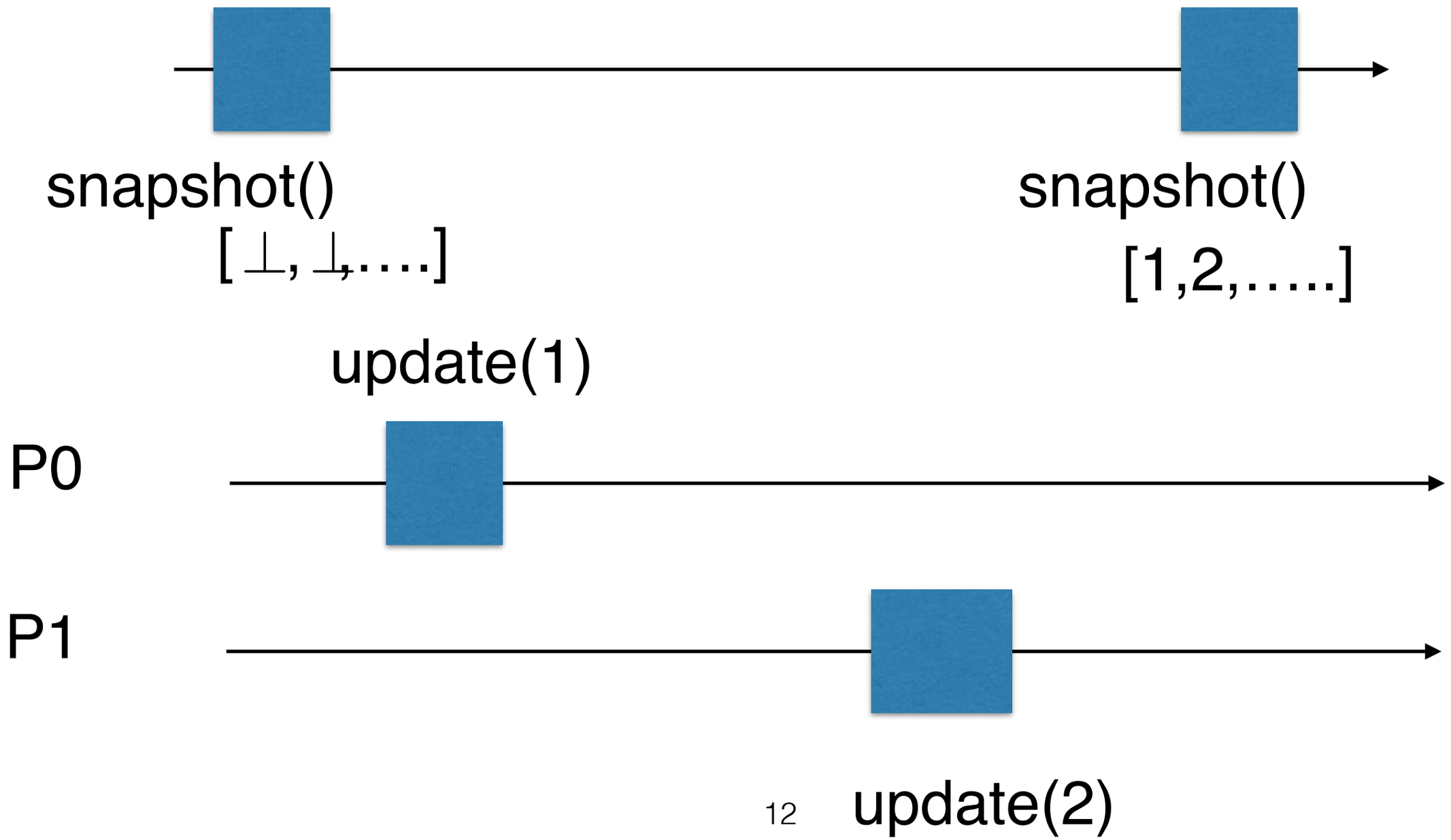

write(1)
ok
write(0) ok
P1
ok
P2
write(2)
read()
0

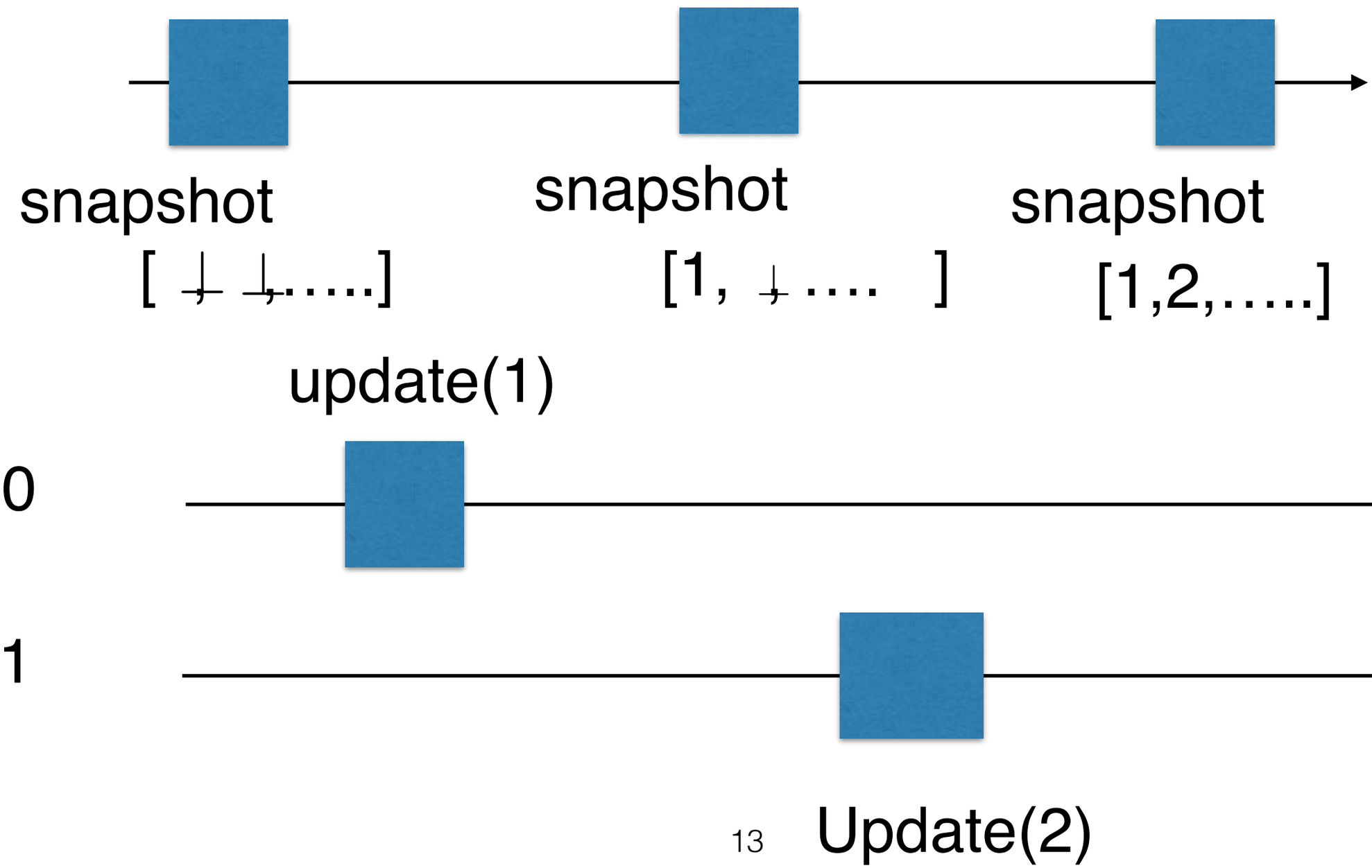
Atomic snapshot

- Goal: reading multiple locations atomically

- Two operations:
    - update(v) return ok
    - snapshot() return an array of n items

- Sequential specification:
    - state: an array A of n items
    - update(v) by process pi writes v in A[i]
    - snapshot () return an array X where X[i] is the last value written by Pi


snapshot()
[⊥, ⊥,….]
snapshot()
[1,2,…..]
update(1)
P0
P1
update(2)


snapshot
snapshot
snapshot
[⊥ ⊥,…..]
[1, ⊥ ….]
[1,2,…..]
update(1)
P0
P1
Update(2)

One update per process

- Property: all the snapshots are ordered

$$S_i \subseteq S_j or S_j \subseteq S_i$$

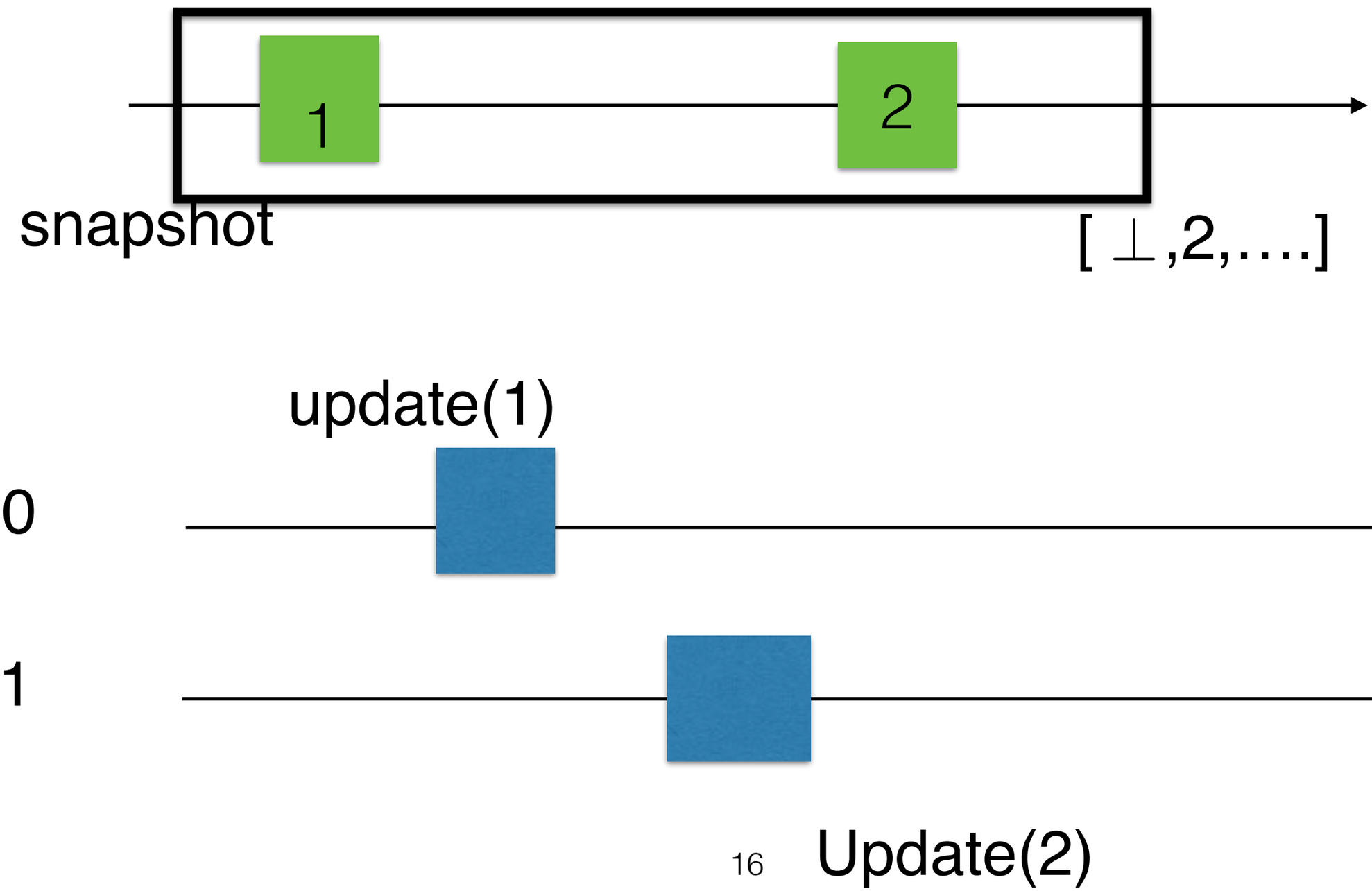
- reduced the complexity of the implementation

first attempt

initially **shared** R array of atomic register init [v,…v]
local to Pi array C of n items init [v,…v]

Code of Pi

```
update(w){
    R.[i]write(w)
    return (ok)
    }
snapshot(){
    for ( int j=0; j<n; j++) C[i]:=R[i].read()
    return C
    }
```


1
2
snapshot
[⊥,2,….]
update(1)
P0
P1
Update(2)

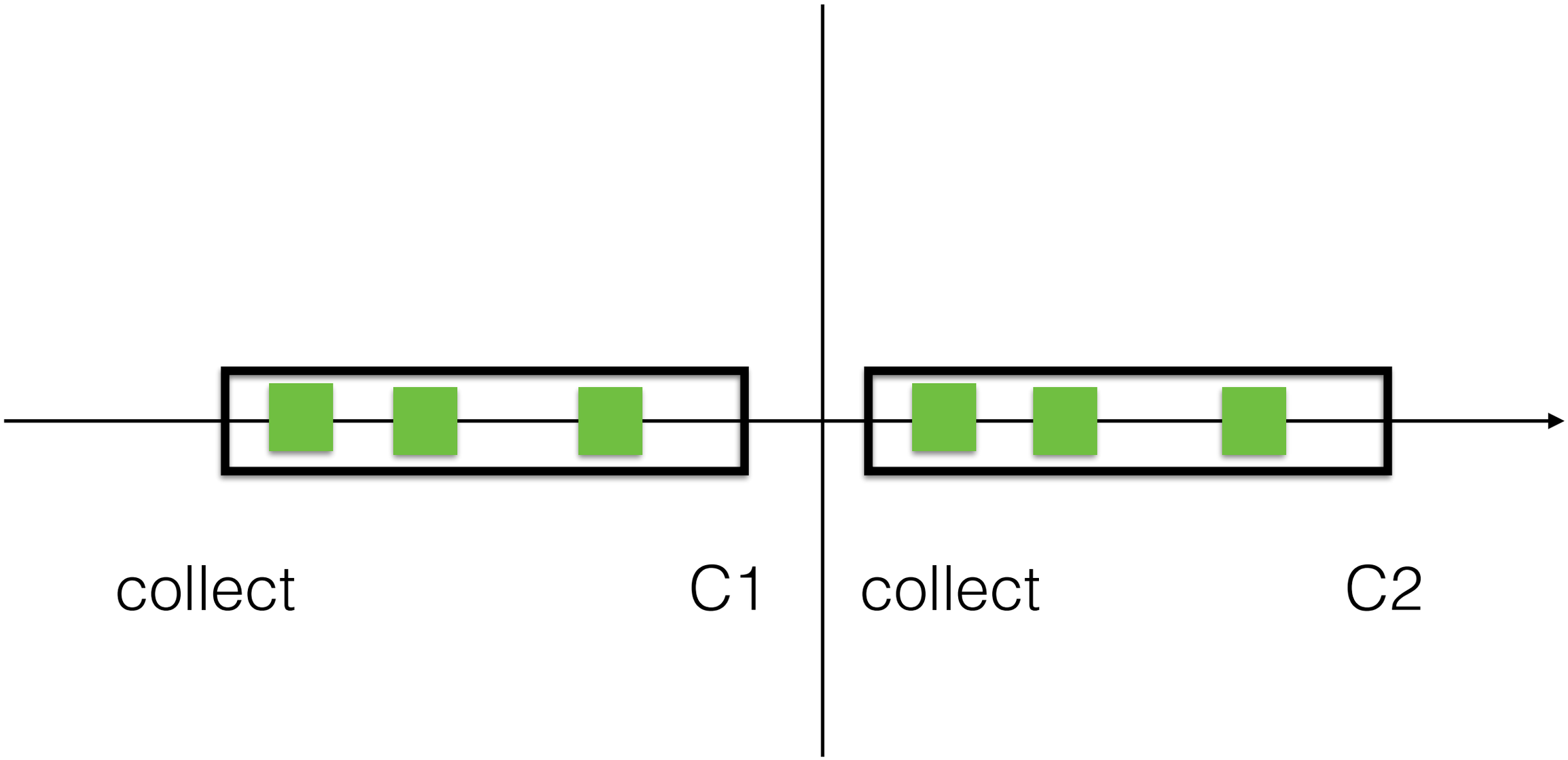
initially **shared** R array of atomic register init [v,…v]
local to Pi array C of n items init [v,…v]

Code of Pi

```
update(w){
    R.[i]write(w)
    return (ok)
    }
collect(){
    for ( int j=0; j<n; j++) C[i]:=R[i].read()
    return C
    }
```

- C1=collect(); C2= collect ()

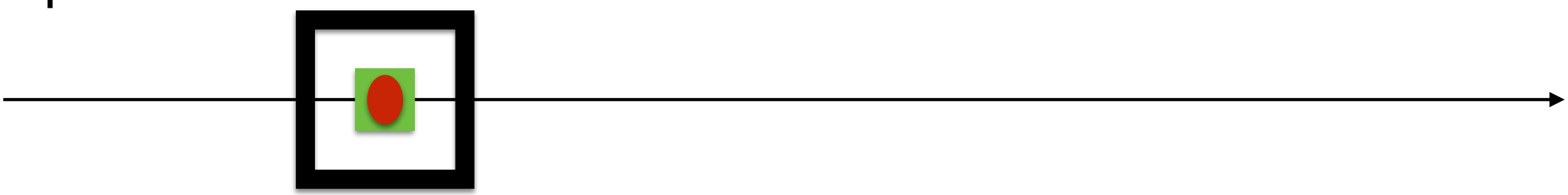
- if C1=C2 then C2 is a snapshot !


collect
C1
collect
C2

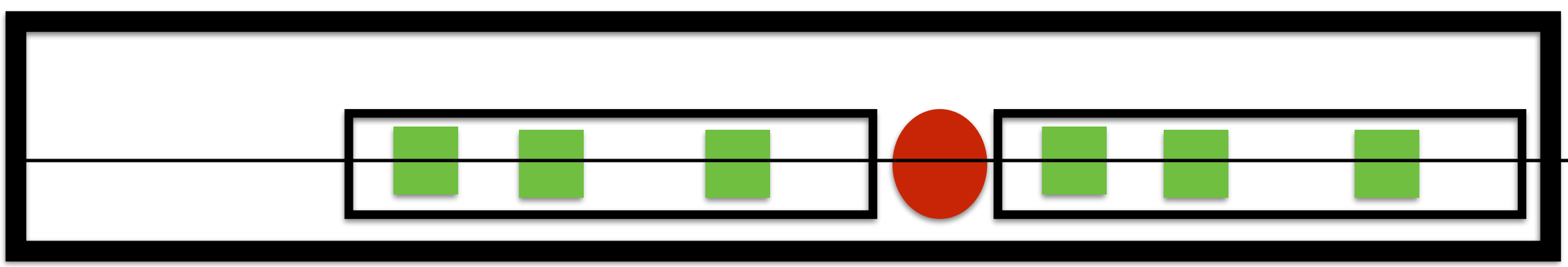
```
initially  shared R  array of atomic register init [v,…v]
           local to Pi array C1,C2  of n items init [v,…v]

Code of Pi
update(w){
     R[i].write(w)
     return (ok)
     }
collect(){
     for ( int j=0; j<n; j++) C[i]:=R[i].read()
     return C
     }
snapshot(){
   C1:=collect()
   forever
        C2:=collect ();
        if (C1==C2) return C1 else C1:=C2
        }
```

Safety : linearization points

update



snapshot C1



collect C1 collect C2

C1==C2

Liveness

- If update() concurrent to collect, snapshot never terminate

- Non blocking

implementation in message passing

- implementation register with crash failures
- Attiya et al.
- a majority of correct process is needed and sufficient

- local variable REG
- write(x): send (write, x) to all
- upon receive (write, x) REG:=x
- read(): return REG

end of write?
what happen if only some process receive (write, x)?
write(x) and write(y), what is the latest ?

- local variable REG,wts,rts
- write(x): wts++;send (write, x,tws) to all; wait (ok_write,ts)from a majority of processes
- upon receive (write, x, t) from p REG:=max(REG, (x,t)) send(ok_write, t) to p

- local variable REG,wts,rst
- read(): send (read, rts) to all; wait (ok_read,v,rts) from a majority of processes; REG=max(REG, v), write(REG)

- upon receive (read, t) from p send(ok_read,REG, t) to p

Implementation

- first registers (write 2n messages; read 4n)
- second snapshot (double collect)
- collect n read —> $4n^2$

code of process i

operation update(v) **is**

(1) $ts_i \leftarrow ts_i + 1$; $reg_i[i] \leftarrow (v, ts_i)$;

(2) broadcast WRITE(reg_i);

(3) **wait** (WRITE_ACK(reg) received from a majority of processes such that $reg_i \preceq reg$);

(4) **let** R be the set reg arrays received at the previous line;

(5) **for** $k \in \{1, \cdots, n\}$ **do**
$reg_i[k] \leftarrow \max_{\preceq}\{r[k] | r \in R \cup reg_i\}$ **end for**;

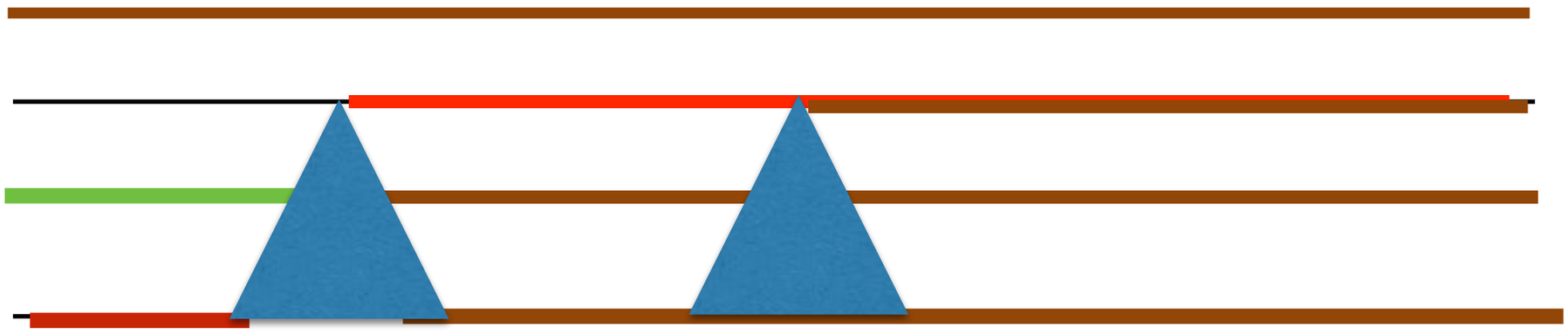
(6) return()

end operation.

operation snapshot() **is**
(7) **repeat**
(8) $prev \leftarrow reg_i$;
(9) $ssn_i \leftarrow ssn_i + 1$; broadcast SNAPSHOT$(reg_i, ssn_i)$;
(10) **wait** (SNAPSHOT_ACK$(-, ssn_i)$ received from a maj. of proc.);
(11) **let** R be the set reg arrays received at the previous line;
(12) **for** $k \in \{1, \cdots, n\}$ **do**
$reg_i[k] \leftarrow \mathsf{max}_{\preceq}\{r[k] | r \in R \cup reg_i\}$ **end for**
(13) **until** $prev = reg_i$ **end repeat**;
(14) return(reg_i)
end operation.

when a message update(reg) **is received from** p_j **do**
(15) **for** $k \in \{1, \cdots n\}$ **do** $reg_i[k] \leftarrow \max_{\preceq}(reg_i[k], reg[k])$ **end for**;
(16) send WRITE_ACK(reg_i) to p_j.

when a message SNAPSHOT(reg, ssn) **is received from** p_j **do**
(17) **for** $k \in \{1, \cdots, n\}$ **do** $reg_i[k] \leftarrow \max_{\preceq}(reg_i[k], reg[k])$ **end for**;
(18) send SNAPSHOT_ACK(reg_i, ssn) to p_j.



Number of messages

- update : 2n messages
- snapshot (good case) : 2n messages

wait free

- helping mechanism

Conclusion

- less messages than
- implementation of registers in MP
- +
- implementation of snapshot in SM