FDS 2012:
Atomic and immediate snapshots
Part II

One-shot atomic snapshot (AS)

Each process p_i:

  $update_i(v_i)$

  S_i := snapshot()

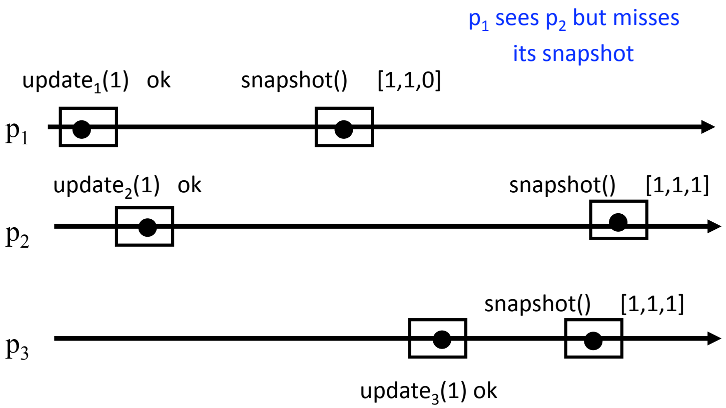
$S_i = S_i[1],\dots,S_i[N]$

(one position per process)

Vectors S_i satisfy:

- Self-inclusion: for all i: v_i is in S_i
- Containment: for all i and j: S_i is subset of S_j or S_j is subset of S_i

"Unbalanced" snapshots



Enumerating possible runs: two processes

Each process p_i (i=1,2):

  $update_i(v_i)$

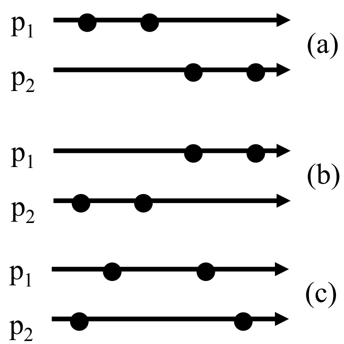
  S_i := snapshot()

Three cases to consider:

(a) p_1 reads before p_2 writes

(b) p_2 reads before p_1 writes

(c) p_1 and p_2 go "lock-step": first both write, then both read

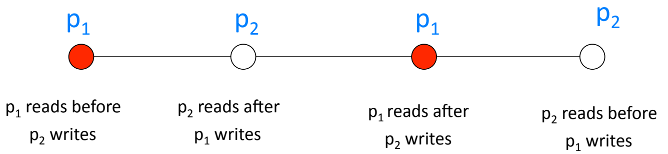


Topological representation of concurrency: two processes

Vertex: local state

Simplex (set of vertexes): global state

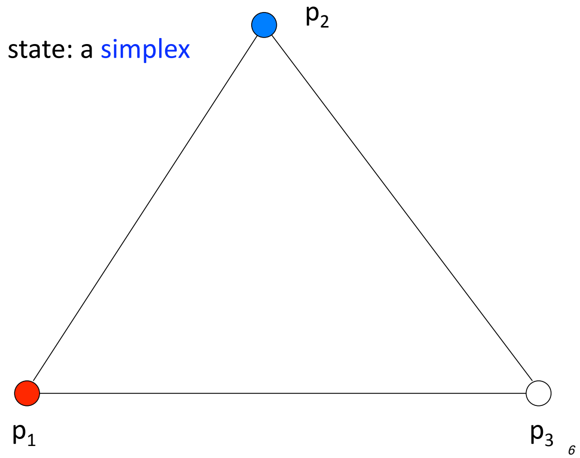
Simplicial complex: set of reachable global states



5

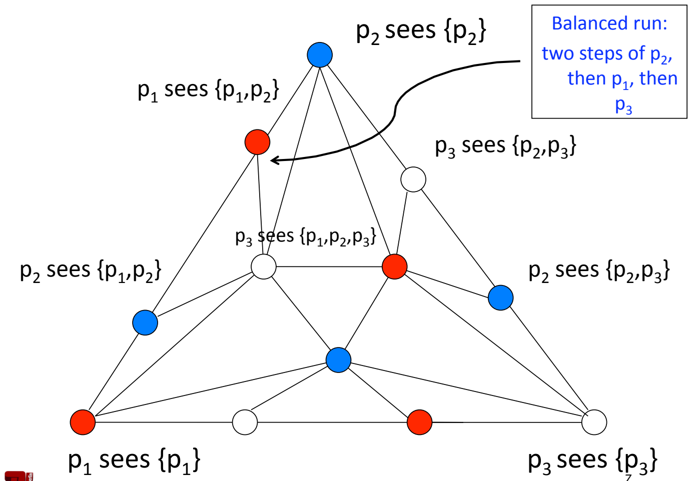
Topological representation: three processes

Initial state: a simplex



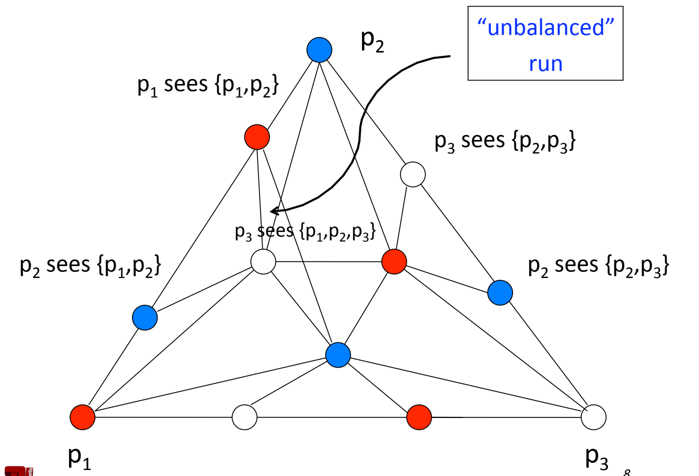
6

Topological representation: one-shot AS



7

Topological representation: one-shot AS



8

One-shot *immediate* snapshot (IS)

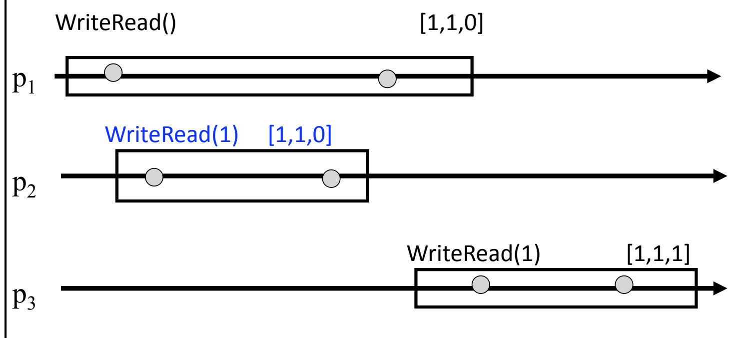
One operation:
WriteRead(v)

Each process p_i:
S_i := $WriteRead_i(v_i)$

Vectors $S_1,\ldots,S_N$ satisfy:

- Self-inclusion: for all i: v_i is in S_i
- Containment: for all i and j: S_i is subset of S_j or S_j is subset of S_i
- Immediacy: for all i and j: if v_i is in S_j, then is S_i is a subset of S_j

"Unbalanced" run eliminated



Every IS run is "balanced"

The runs of IS match the block runs of one-shot AS that can be written as:

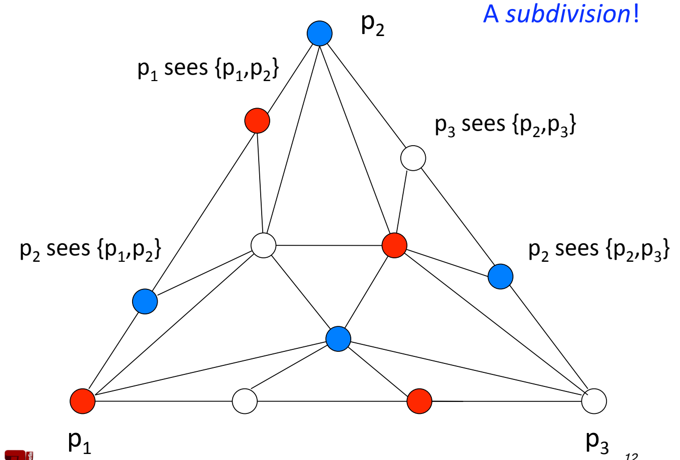
$$B_1\ B_2 \ldots B_k$$

where

- each B_i = $update_{\Pi i}()$ $snapshot_{\Pi i}()$
  - ✓All processes in Π_i update and then all processes in Pi take snapshots
- $\Pi_1,\ldots,\Pi_k$ is a partition of $\{p_1,\ldots,p_N\}$
  - ✓E.g., $\Pi_1=\{p_2\}$; $\Pi_2=\{p_1\}$; $\Pi_3=\{p_3\}$ in the balanced run above

Topological representation: one-shot IS

A subdivision!



IS is equivalent to AS (one-shot)

- IS is a restriction of one-shot AS => IS is stronger than one-shot AS
  - ✓Every run if IS is a run of one-shot AS
- Show that a few (one-shot) AS objects can be used to implements IS
  - ✓One-shot ReadWrite() can be implemented using a series of update and snapshot operations

IS from AS

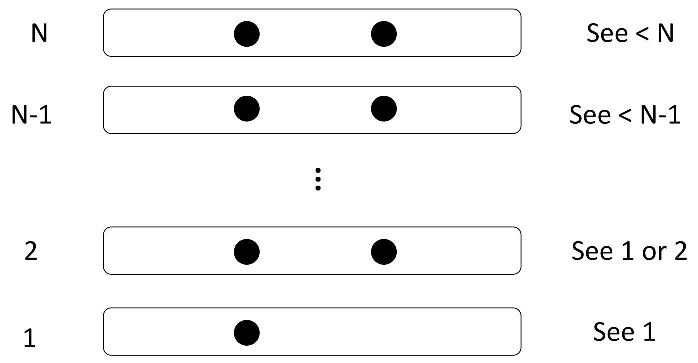
shared variables:

$A_1,\ldots,A_N$ – atomic snapshot objects, initially [T,…,T]

Upon WriteRead$_i$(v_i)

```
r := N+1
while true do
        r := r-1            // drop to the lower level
        A_r.update_i(v_i)
        S := A_r.snapshot()
        if |S|=r then       // |S| is the number of non-T values in S
            return S
```


Drop levels: two processes, N>3



Correctness

The outcome of the algorithm satisfies Self-Inclusion, Snapshot, and Immediacy

- By induction on N: for all N>1, if the algorithm is correct for N-1, then it is correct for N
- Base case N=1: trivial

Correctness, contd.

- Suppose the algorithm is correct for N-1 processes
- N processes come to level N
  - ✓At most N-1 go to level N-1 or lower
  - ✓(At least one process returns in level N)
  - ✓Why?
- Self-inclusion, Containment and Immediacy hold for all processes that return in levels N-1 or lower
- The processes returning at level N return all N values
  - ✓The properties hold for all N processes! Why?

Iterated Immediate Snapshot (IIS)

Shared variables:

IS_1, IS_2, IS_3,… // a series of one-shot IS

Each process p_i with input v_i:

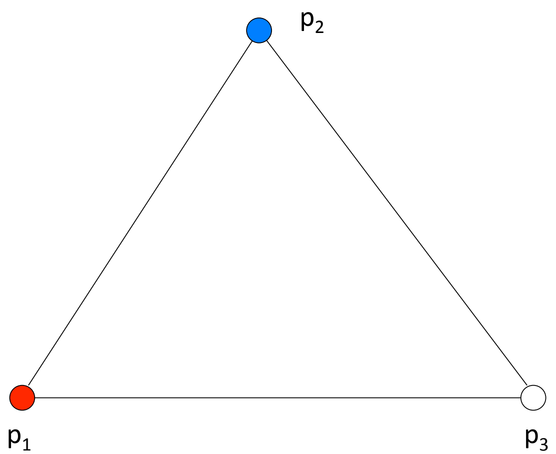
r := 0

while true do

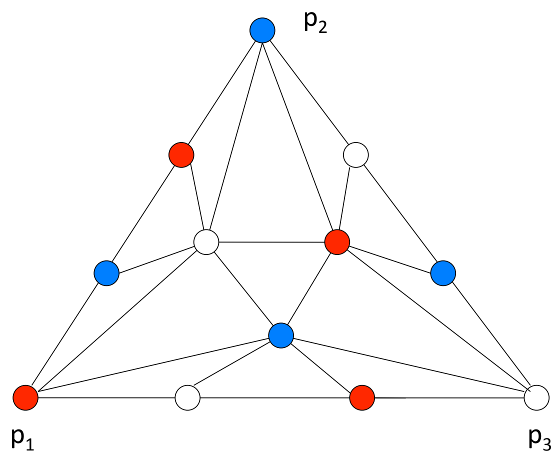
r := r+1

v_i := IS_r.WriteRead$_i$(v_i)

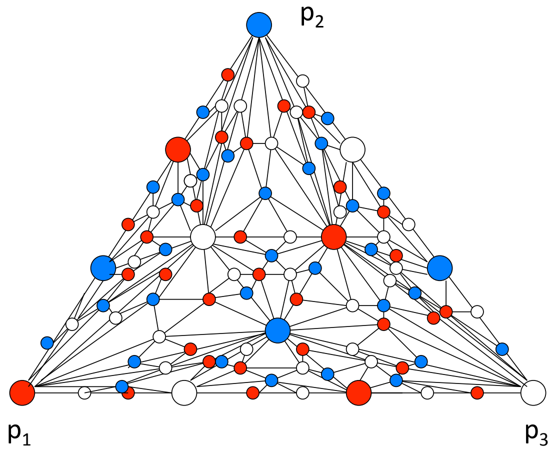
Iterated standard chromatic subdivision (ISDS)



ISDS: one round of IIS



ISDS: two rounds of IIS



IIS is equivalent to (multi-shot) AS

- AS can be used to implement IIS (wait-free)
  - ✓Multiple instances of the construction above (one per iteration)
- IIS can be used to implement multi-shot AS in the non-blocking manner:
  - ✓At least one correct process performs infinitely many read or write operations
  - ✓Good enough for protocols solving distributed tasks!

From IIS to AS

We simulate an execution of full-information protocol (FIP) in the AS model, i.e., each process p_i runs:

state := input value of p_i
repeat
  update$_i$(state)
  state := snapshot()
until undecided(state)

Recursively, vector of vectors

(the input value and the decision procedure depend on the problem being solved)

If a problem is solvable in AS, it is solvable with FIP

From IIS to AS

Each process p_i maintains a vector clock c[1,…,N]

- Each c[j] has two components:
  - ✓c[j].clock: the number of updates of p_j "witnessed" by p_i
  - ✓c[j].val: the most recent value of p_j's vector clock "witnessed" by p_i
- To perform an update: increment c[i].clock and set c[i].val to be the "most recent" vector clock
- To take a snapshot: go through iterated memories until $|c| = \Sigma_j c[j].clock$ is "large enough"

- $c \geq c'$ iff for all j, c[j].clock ≥ c'[j].clock (c observes a more recent state than c)
- $|c| = \Sigma_j$c[j].clock (sum of clock values of the last seen values)
- For c = c[1],...c[N] (vector of vectors c[j]), top(c) is the vector of most recent seen values:

c[1]	= [(1,a)	(3,b)	(2,c)]	
c[2]	= [(4,u)	(2,v)	(1,w)]	
c[3]	= [(2,x)	(1,y)	(5,z)]	
top(c)	= [(4,u)	(3,b)	(5,z)]	

From IIS to AS: non-blocking simulation

Shared: $IS_1, IS_2, \ldots$ // an infinite sequence of one-shot IS memories
Local: at each process, c[1,...,N]=[(0,T),...,(0,T)]
Code for process p_i:

```
r:=0; c[i].clock:=1; c[i].val := input of p_i // register p_i's input
repeat forever
        r:=r+1
        view := IS_r.WriteRead(c) // get the view in IS_r
        c := top(view) // get reported clock per process
        if |c|=r then //the current snapshot completed
           if undecided(c.val) then
                 c[i].val:=c.val // update the value
                 c[i].clock:=c[i].clock+1 // update the clock
           else
                 return decision(c.val) // return the decision
```


From IIS to AS: correctness

Let c_r denote the vector evaluated by an undecided process p_i in round r (after computing the top function)

Lemma 1 $|c_r| \geq r$

Proof sketch

$c_r \geq c_{r+1}$ (by the definition of top)

Initially $|c_1| \geq 1$ (each process writes c[1].clock=1 in IS_1)

Inductively, suppose $|c_r| \geq r$, for some round r:

- If $|c_r|=r$, then c', $|c'|=r+1$, is written in IS_{r+1}
- If $|c_r|>r$, then c', such that $c' \geq c_r$ (and thus $|c'| \geq |c_r|$) is written in IS_{r+1}

In both cases, $c_{r+1} \geq r+1$

From IIS to AS: correctness

Lemma 2 Let c_r and c_r' be the clock vectors evaluated by processes p_i and p_j, resp., in round r. Then $|c_r| \leq |c_r'|$ implies $c_r \leq c_r'$

Proof sketch

Let S_i and S_j be the outcomes of IS_r received by p_i and p_j

$c_r = top(S_i)$ and $c_r' = top(S_j)$

Either S_i is a subset of S_j or S_j is a subset of S_i (the Containment property of IS)

Suppose S_i is a subset of S_j, then for each clock value seen by p_i is also seen by p_j Why?

=> $|c_r| \leq |c_r'|$ and $c_r \leq c_r'$ Why?

Corollary 1 (to Lemma 2) All processes that complete a snapshot operation in round r, get the same clock vector c, $|c|=r$

Corollary 2 (to Lemmas 1 and 2) If a process completes a snapshot operation in round r with clock vector c, then for each clock vector c' evaluated in round r '≥r, we have $c_r \leq c_r'$

From IIS to AS: linearization

Lemma 3 Every execution's history is linearizable (with respect to the AS spec.)

Proof sketch

Linearization

- Order snapshots based on the rounds in which they complete
- Put each update(c) just before the first snapshot that contains c (if no such snapshot – remove)

By Corollaries 1 and 2, snapshots and updates in the order respect the specification of AS - legality

Both linearization points take place "within the interval" of k-th update and k-th snapshot of p_i - between k-th and (k+1)-th updates of c[i].val – precedence

From IIS to AS: liveness

Lemma 4 Some correct undecided process completes infinitely many snapshot operations (or every process decides).

Proof sketch

By Lemma 1, a correct process p_i does not complete its snapshot in round r only if $|c_r|>r$

Suppose p_i never completes its snapshot

=> c_r keeps grows without bound and

=> some process p_j keeps updating its c[j]

=> some process p_j completes infinitely many snapshots

IIS=AS for wait-free task solutions

- Suppose we simulate a wait-free protocol for solving a task:
  - ✓Every process starts with an input
  - ✓Every process taking sufficiently many steps (of the full-information protocol) eventually decides (and thus stops writing new values, but keeps writing the last one)
  - ✓Outputs match inputs (we'll see later how it is defined)
- If a task can be solved in AS, then it can be solved in IIS
  - ✓We consider IIS from this point on

Homework 4

4.1 Complete the proof of the one-shot IS algorithm

4.2 Would the algorithm be correct if we replace A_r.update$_i$(v_i) with U_r[i].write(v_i) and A_r.snapshot() with scan(U_r[1],…,U_r[N])?

4.3 Complete the proofs of Lemma 2 and Corollaries 1 and 2