

# Task

# tasks...

As « task »:  $\mathcal{T} = \langle I, O, R \rangle$

- Input  $I$ : vector, value  $v$  for each process
- Output  $O$ : value for each process.
- $R$  relation on  $I \times O$

Wait-free task:

- in the input vector  $\perp$  denotes a process that does not make any step.
- In the output vector  $\perp$  denotes a no decision (process has crashed)
- Wait-free: we can replace input by  $\perp$  if process is crashed

## Consensus for 2 process

$(1,0) (0,1)(1,1) \rightarrow (1,1) (1, \text{bot}) ( \text{bot},1)$

$(1,0) (0,1)(0,0) \rightarrow (0,0) (0, \text{bot}) ( \text{bot},0)$

$(1, \text{bot}) \rightarrow (1, \text{bot}) ; ( \text{bot},1) \rightarrow ( \text{bot},1)$

$(0, \text{bot}) \rightarrow (0, \text{bot}) ; ( \text{bot},0) \rightarrow ( \text{bot},0)$

Immediate snapshots  
Iterated Immediate snapshot

# One-shot atomic snapshot (AS)

Each process  $p_i$ :

update $_i(v_i)$

$S_i := \text{snapshot}()$

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

(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  $S_i$

# Enumerating possible runs: two processes

Each process  $p_i$  ( $i=1,2$ ):

$update_i(v_i)$

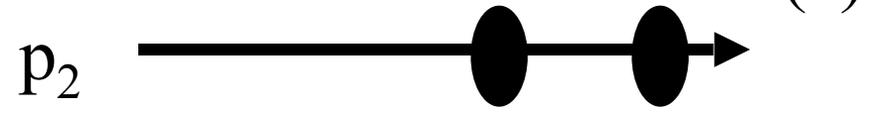
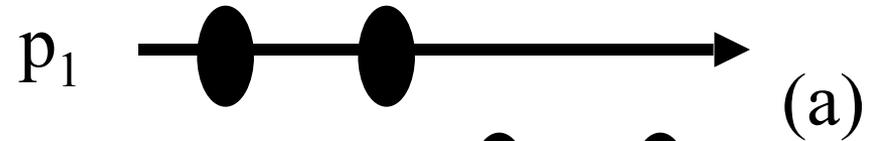
$S_i := snapshot()$

Three cases to consider:

(a)  $p_1$  reads before  $p_2$  writes

(b)  $p_2$  reads before  $p_1$  writes

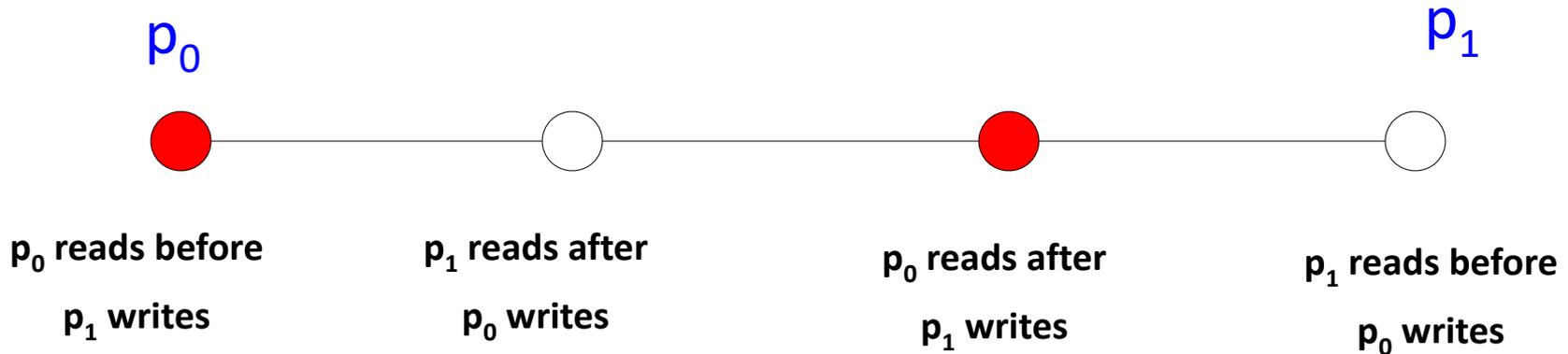
(c)  $p_1$  and  $p_2$  go “lock-step”: first both write, then both read



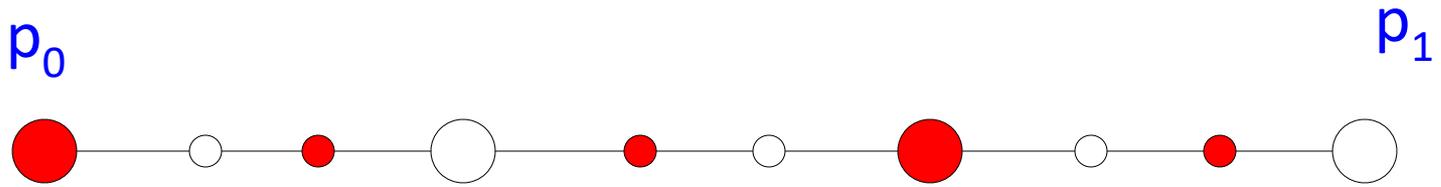
$n=2$

Initially each  $p_i$  only knows its input

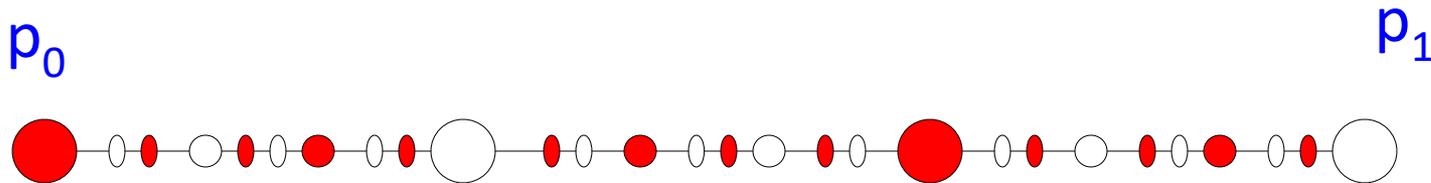
One round of write-read ( idem write-snapshot):



Two rounds:



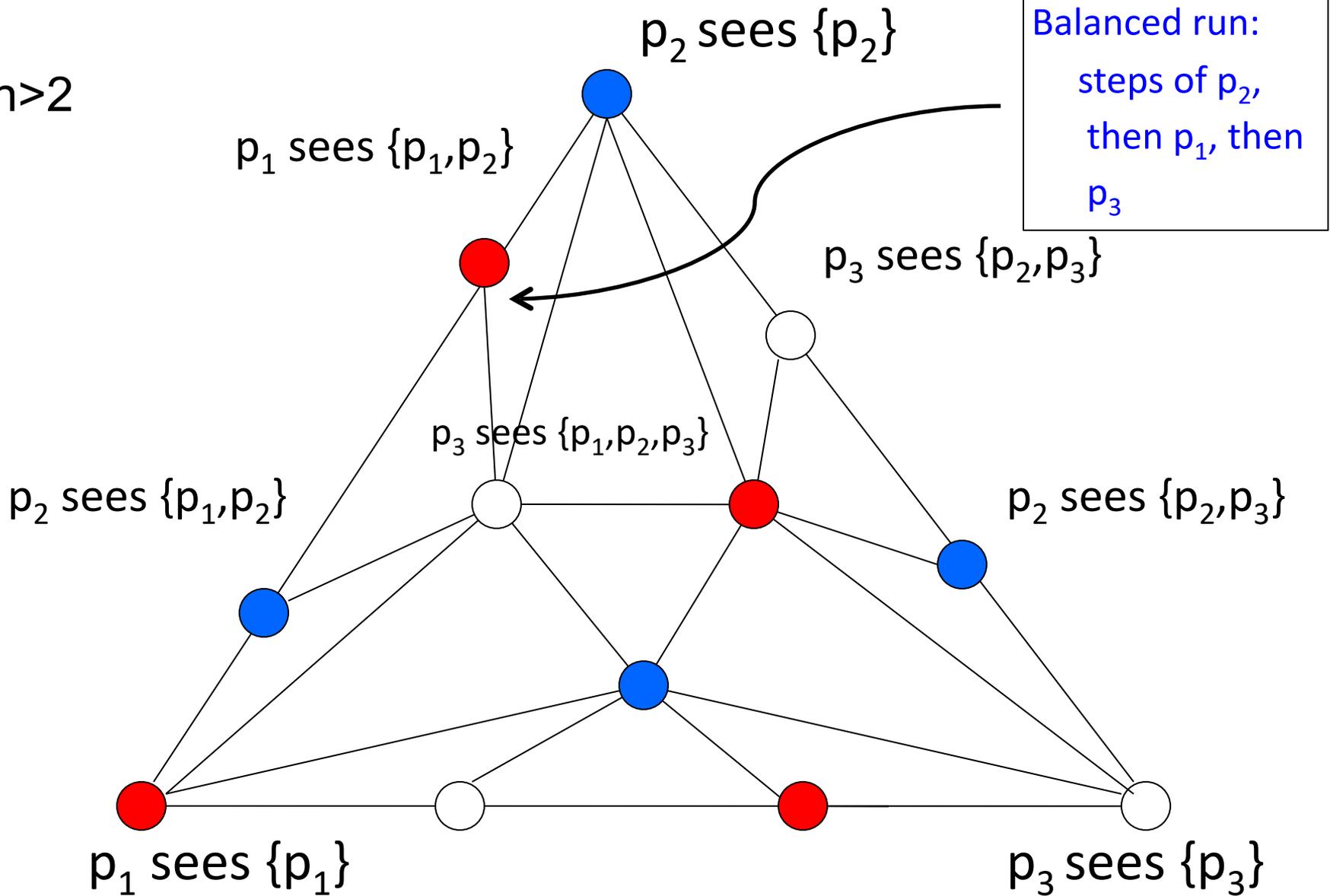
And so on...



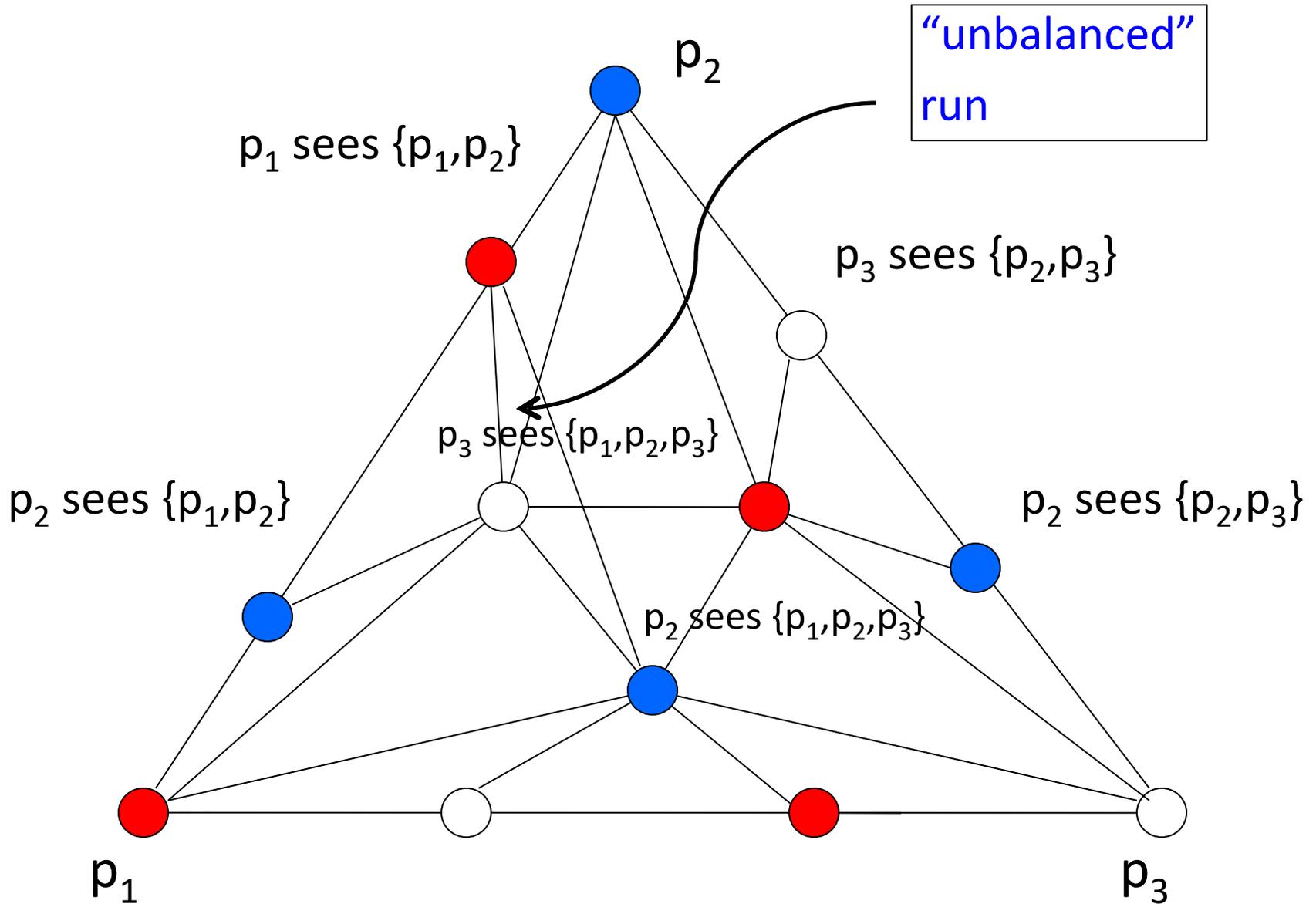
In this iterated model : we prove the  
impossibility of wait free consensus for  
two processes

# Topological representation: one-shot AS

$n > 2$



# Topological representation: one-shot AS



# One-shot *immediate* snapshot (IS)

One operation:  
WriteRead( $v$ )

Each process  $p_i$ :

$S_i := \text{WriteRead}_i(v_i)$

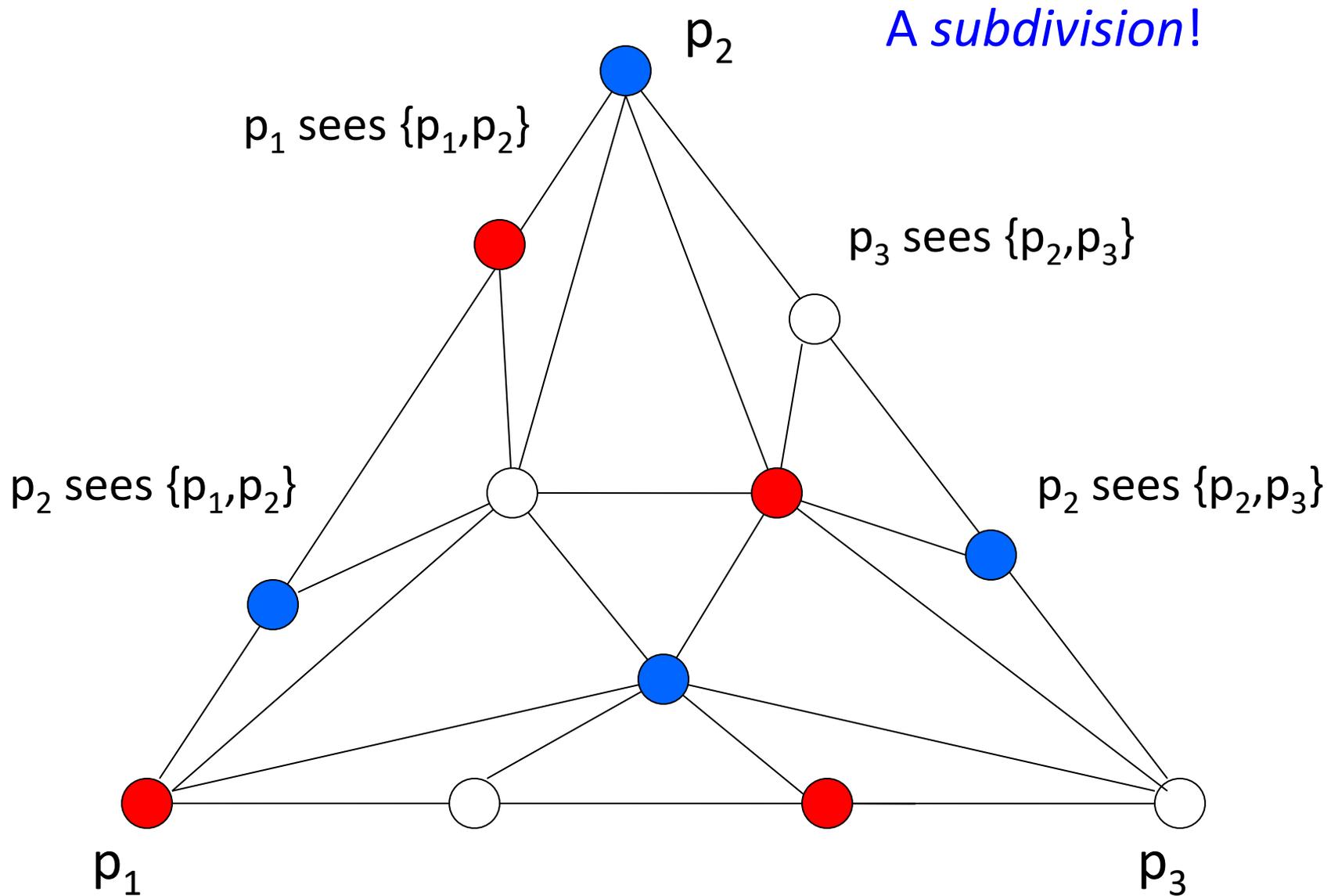
Vectors  $S_1, \dots, S_N$  satisfy:

§ **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  $S_i$  is a subset of  $S_j$

# Topological representation: one-shot IS



# IS is equivalent to AS (one-shot)

- IS is a **restriction** of one-shot AS  $\Rightarrow$  IS is **stronger** than one-shot AS
  - ✓ Every run of 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:

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

## Upon WriteRead<sub>i</sub>( $v_i$ )

$r := N+1$

while true do

$r := r-1$                    // drop to the lower level

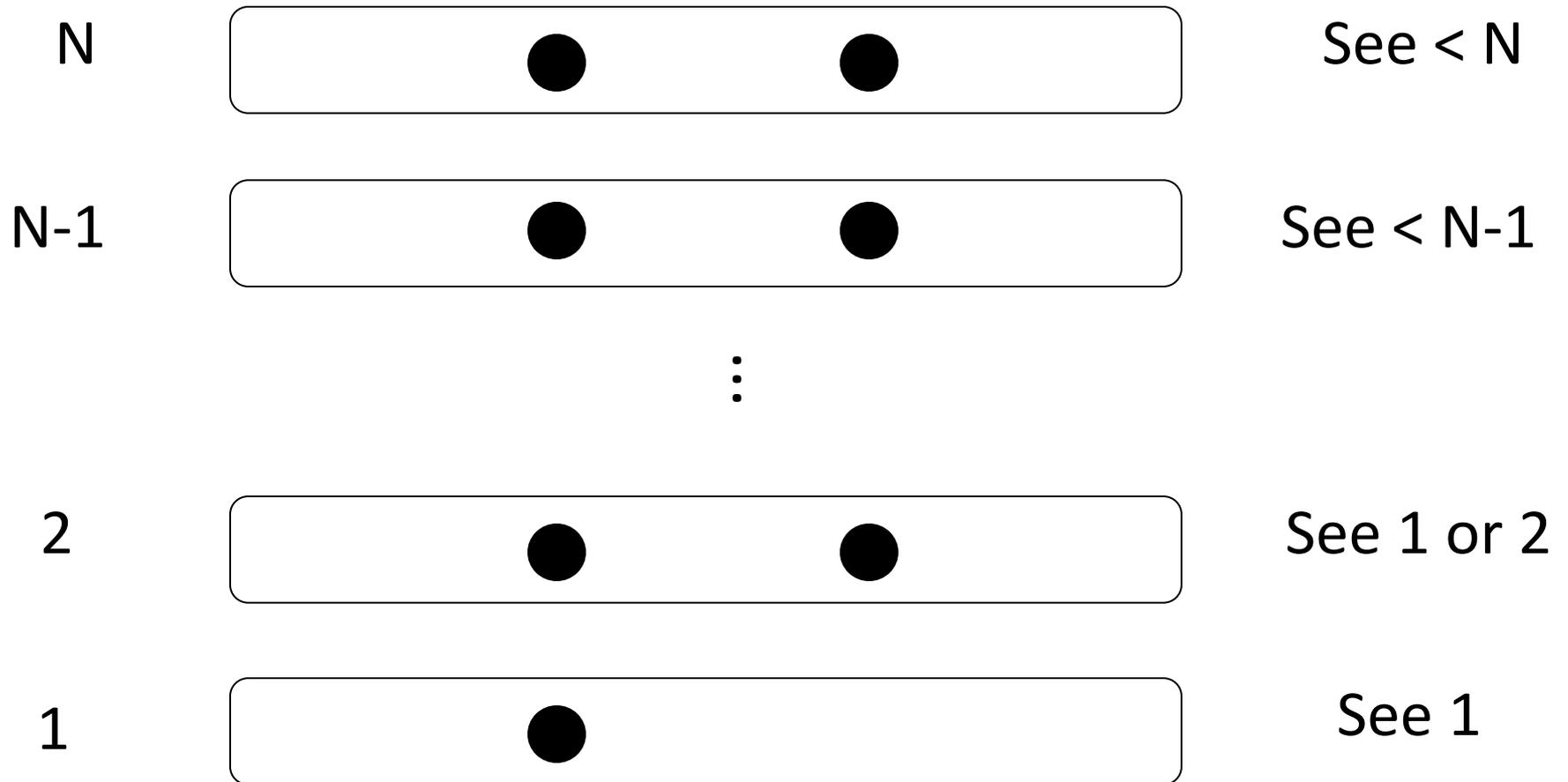
$A_r.update_i(v_i)$

$S := A_r.snapshot()$

    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  $N$
- 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:

$IS_1, IS_2, IS_3, \dots$  // a series of one-shot IS

Each process  $p_i$  with input  $v_i$ :

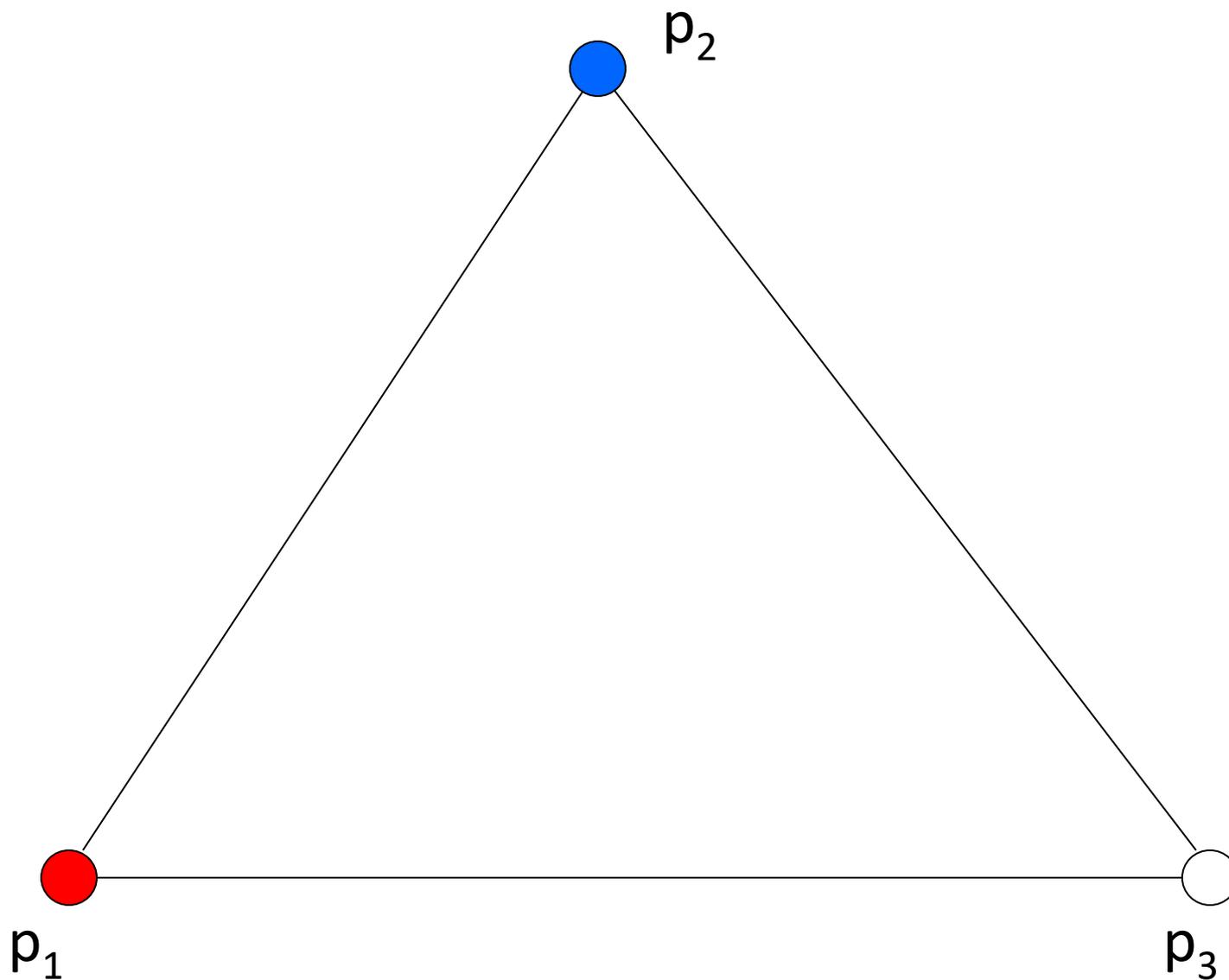
$r := 0$

while true do

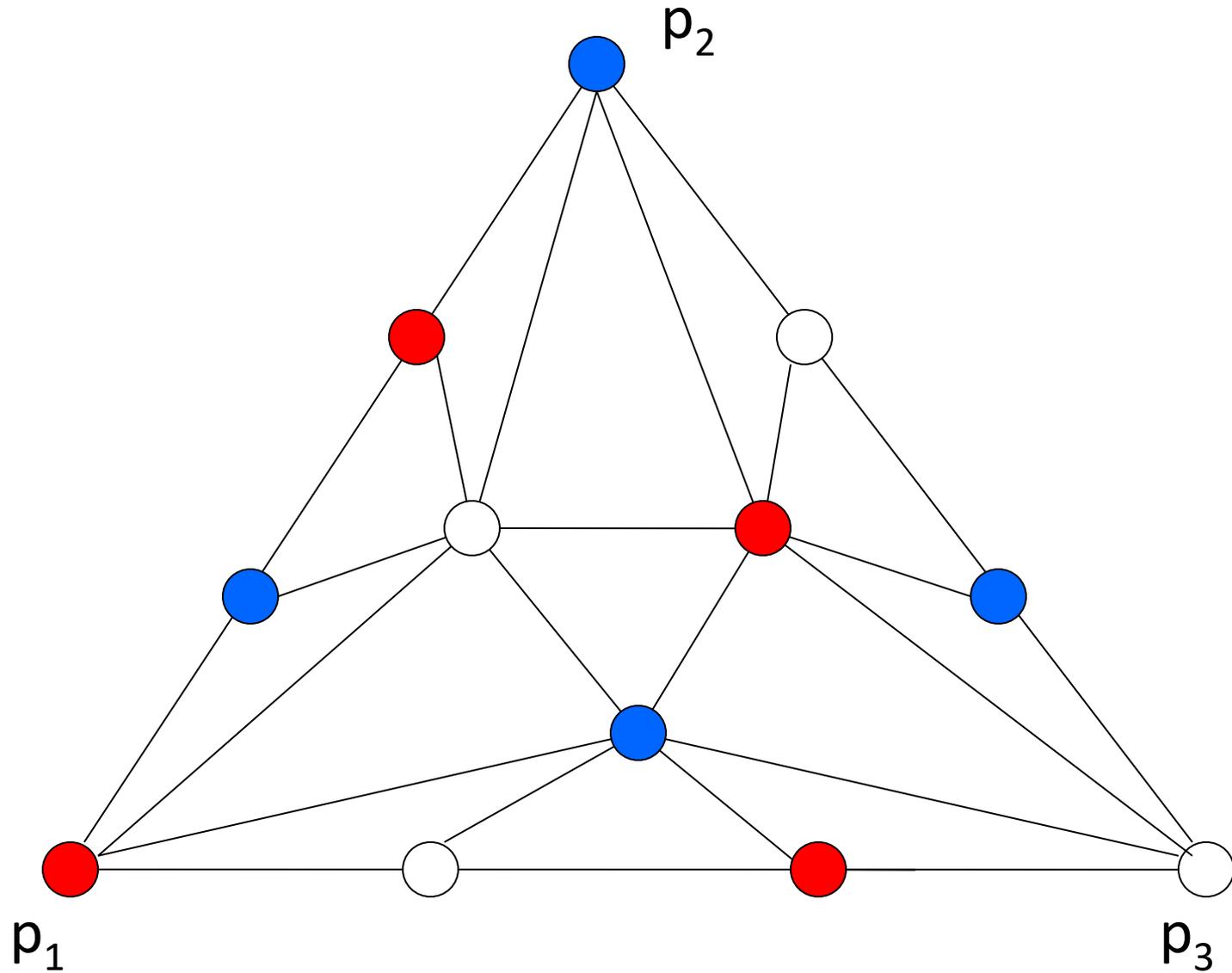
$r := r + 1$

$v_i := IS_r.\text{WriteRead}_i(v_i)$

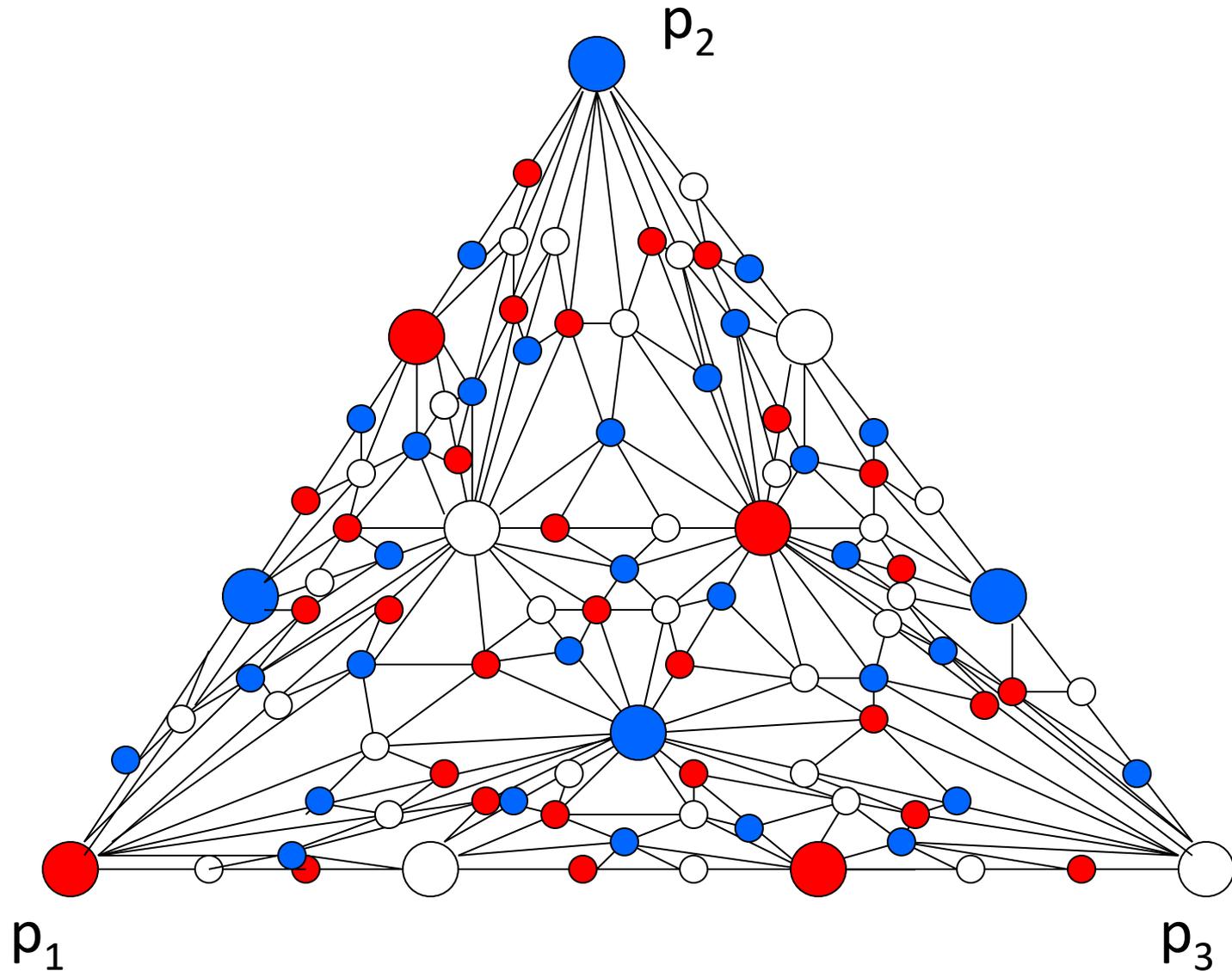
# Iterated standard chromatic subdivision (ISDS)



# 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 decided(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**

**For simplicity, assume that the  $k$ -th written value =  $k$**   
**(“without loss of generality” – every written value is unique)**

# From IIS to AS: non-blocking simulation

**Shared:**  $IS_1, IS_2, \dots$  // an infinite sequence of one-shot IS memories

**Local:** at each process,  $c[1, \dots, N] = [(0, T), \dots, (0, T)]$

**Code for process  $p_i$ :**

$r := 0; c[i].clock := 1; // p_i$ 's initial value

repeat forever

$r := r + 1$

    view :=  $IS_r$ .WriteRead(c) // get the view in  $IS_r$

    c := top(view) // get the top clock values

    if  $|c| = r$  then // the current snapshot completed

        if undecided(ctop) then

$c[i].val := ctop;$

$c[i].clock := c[i].clock + 1 // update the clock$

        else

            return decision(ctop) // return the decision

# From IIS to AS

Each process  $p_i$  maintains a **vector clock**  $c[1, \dots, N]$

- Each  $c[j]$  has two components:
  - ✓  **$c[j].clock$** : the number of updates of  $p_j$  “witnessed” by  $p_i$  ( $c.clock$  - the corresponding vector)
  - ✓  **$c[j].val$** : the most recent value of  $p_j$ ’s vector clock “witnessed” by  $p_i$  ( $c.val$  – the corresponding vector)
- 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  $lcl = \sum_j c[j].clock$  is “large enough”,
  - ✓ i.e.  $lcl = r$  (the current round number)
  - ✓ **As we’ll see,  $lcl \geq r$ : every process  $p_i$  begins with  $c[i]=1$**

- We say that  $c \geq c'$  iff for all  $j$ ,  $c[j].\text{clock} \geq c'[j].\text{clock}$  ( $c$  observes a **more recent** state than  $c'$ )
  - ✓ Not always the case with  $c$  and  $c'$  of different processes
- $|c| = \sum_j c[j].\text{clock}$  (sum of clock values of the last **seen** values)
- For  $c = c[1], \dots, c[N]$  (vector of vectors  $c[j]$ ), **top(c)** is the vector of most recent seen values:

$$\begin{array}{lcl}
 c[1] & = & [1 \quad 3 \quad 2] \\
 c[2] & = & [4 \quad 2 \quad 1] \\
 c[3] & = & [2 \quad 1 \quad 5] \\
 \\ 
 \text{top}(c) & = & [4 \quad 3 \quad 5]
 \end{array}$$

# 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**

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

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

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

- If  $|c_r| = r$ , then  $c'$ , such that  $|c'| = r+1$ , 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  $p_j$

$$c_r = \text{top}(S_i) \text{ and } c_r' = \text{top}(S_j)$$

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 each clock value seen by  $p_i$  is also seen by  $p_j$

$$\Rightarrow |c_r| \leq |c_r'| \text{ and } c_r \leq c_r'$$

# From IIS to AS: correctness

**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  $r' \geq r$ , we have  $c \leq c'$

# 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 put in this order respect the specification of AS

# 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

$\Rightarrow c_r$  keeps grows without bound and

$\Rightarrow$  **some** process  $p_j$  keeps updating its  $c[j]$

$\Rightarrow$  **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