Universality

- A class C of objects is universal for some set E of classes if and only if any object in E can be implemented with objects of C (and registers)

n-Consensus is universal for n-objects (objects shared by n processes)

- for any sequential object there is a wait-free (linearizable) implementation of this object with atomic registers and n-consensus objects.

Universal Construction

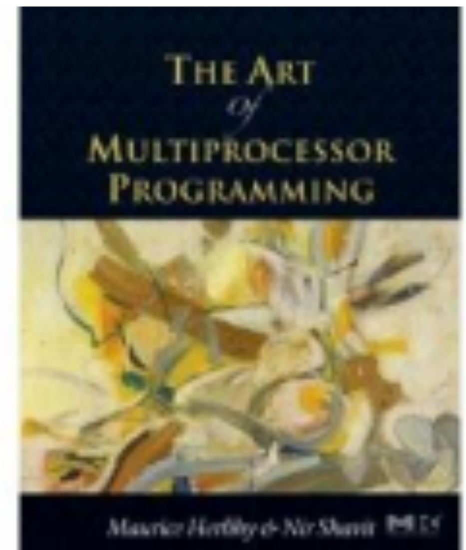
Theorem [Herlihy, 1991] If N processes can solve consensus, then N processes can (wait-free) implement every object O=(Q,O,R,σ) (shared by N processes

Universality

- Principles:
  - build a shared list corresponding to the history of the object (each cell contain the operation and a link to the next cell)
  - link new cells by consensus (hence there is one list!):
    - if a process wants to add a new operation, it goes to the head of the list and proposes the operation to the consensus
  - to ensure wait-freedom a « helping mechanism » (processes propose operations of other processes) (recall that with a consensus object some processes may loose forever)

Universality

- an implementation (Herlihy-Shavit) :



en java

```
public interface SeqObject {
  public abstract
    Response apply(Invoc invoc);
}
```

```
public class Invoc {
 public String method;
 public Object[] args;
}
```

```
public class Response {
  public Object value;
}
```

Implementation

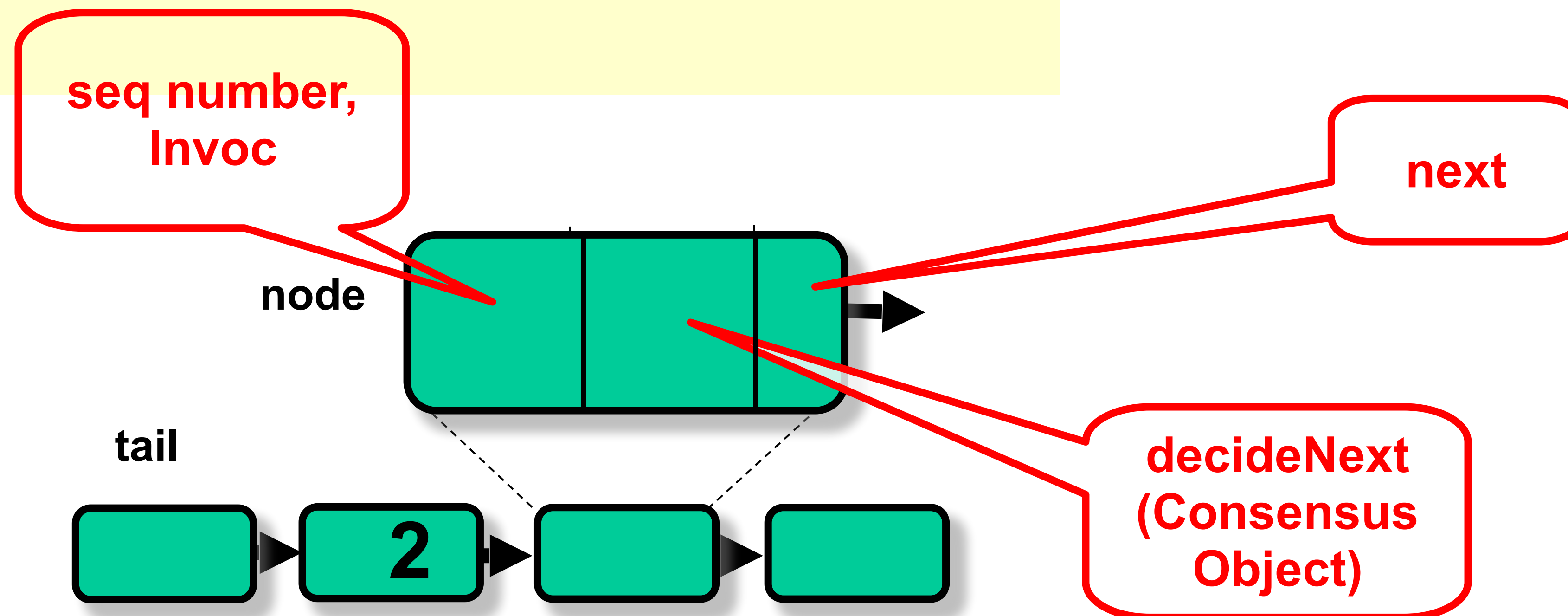
- First: lock-free implementation (infinitely often some method call finishes)
- Then wait-free: each method call takes a finite number of steps to finish

- Object represented as
  - Initial Object State
  - A Log: a linked list of the method calls
- New method call
  - Find end of list
  - Atomically append call (using consensus)
  - Compute response by replaying log
- Threads use *one-time* consensus objects to decide which node goes next
- Threads update actual next field to reflect consensus outcome
  - OK because they all write the same value

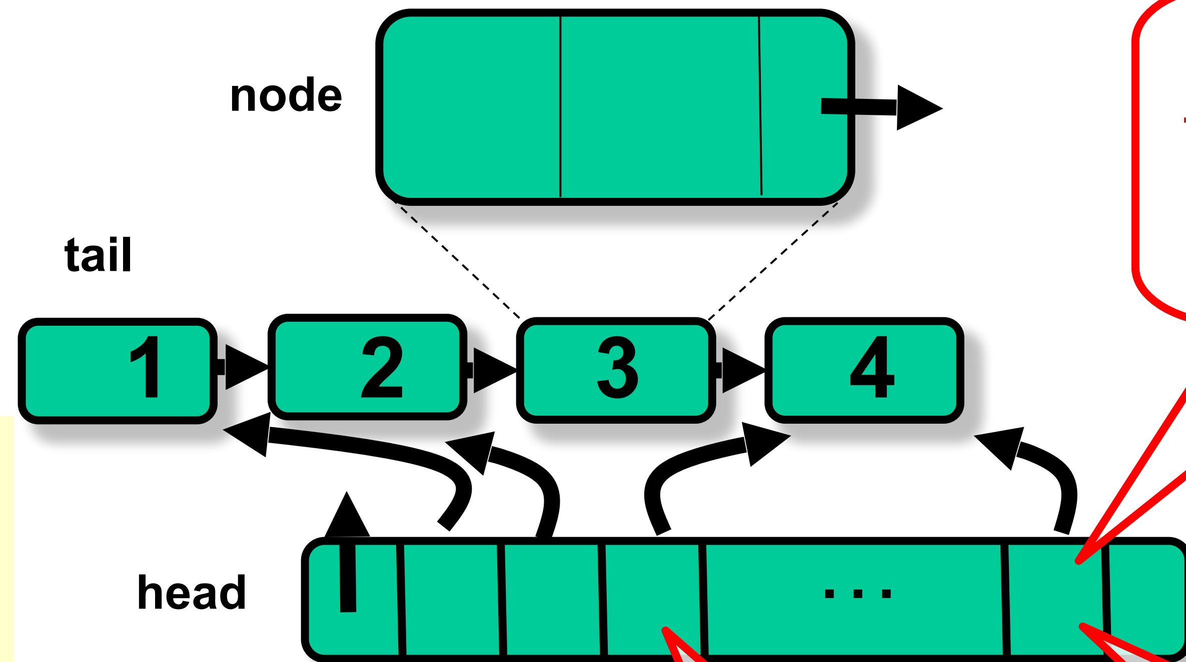
Data Structure

```
public class Node implements
java.lang.Comparable {
 public Invoc invoc;
 public Consensus<Node> decideNext;
 public Node next;
 public int seq;
 public Node(Invoc invoc) {
    invoc = invoc;
    decideNext = new Consensus<Node>()
    seq = 0;
  }
```

warning: in a one-shot consensus a process can do at most one consensus per Node



add the node to the head: how to maintain the head array ?




```
public class Universal{
  private Node[] head;
  public Universal(){
   private Node tail = new
            Node();
   tail.seq = 1;
   for (int j=0; j < n; j++){
         head[j] = tail;
   }
   …
```

```
public static Node max(Node[] array) {
    Node max = array[0];
    for (int i = 1; i < array.length; i++)
      if (max.seq < array[i].seq)
        max = array[i];
      return max;
  }
```

Link the invoc

```
public Response apply(Invoc invoc) {
  int i = myID();
  Node prefer = new node(invoc);
  while (prefer.seq == 0) {
    Node before = Node.max(head);
    Node after =
     before.decideNext.decide(prefer);
    before.next = after;
    after.seq = before.seq + 1;
    head[i] = after;
    }
  …
```

propose prefer to consensus

Compute response

```
...
//compute my response
SeqObject MyObject = new SeqObject();
current = tail.next;
while (current != prefer){
   MyObject.apply(current.invoc);
   current = current.next;
  }
return MyObject.apply(current.invoc);
}
```

Correctness

- List defines linearized sequential history
- Thread returns its response based on list order

- Lock-free because
  - A thread moves forward in list
  - Can repeatedly fail to win consensus on “real” head only if another succeeds
  - Consensus winner adds node and completes within a finite number of steps

wait-free

- not wait-free: if concurrent infinitely often with other invocations an invocation may be forever not chosen in consensus.
- solution helping
  - Lock-free construction + `announce` array
  - Stores (pointer to) node in `announce`
    - If a thread doesn't append its node
    - Another thread will see it in array and ***help*** append it

```
public class Universal {
  private Node[] announce;
  private Node[] head;
  public Universal(){
     private Node tail = new node();
     tail.seq = 1;
     for (int j=0; j < n; j++){
       head[j] = tail; announce[j] = tail
     };
```

announce array

```
public Response apply(Invoc invoc) {
  int i = ThreadID.get();
  announce[i] = new Node(invoc);
  head[i] = Node.max(head);
  while (announce[i].seq == 0) {
  …
  // while node not appended to list
  …
  }
```

help!

- Non-zero sequence # means success
- Thread keeps helping append nodes
- Until its own node is appended

```
while (announce[i].seq == 0) {
 Node before = head[i];
 Node help = announce[(before.seq + 1) % n];
 if (help.seq == 0)
     prefer = help;
    else
     prefer = announce[i];
…
```

- Choose a thread to “help”
- If that thread needs help
  - Try to append its node
  - Otherwise append your own
- Worst case
  - Everyone tries to help same pitiful loser
  - Someone succeeds

- When last node in list has sequence number *k*
- All threads check …
  - Whether thread *k*+1 mod *n* wants help
  - If so, try to append her node first

- First time after thread *k*+1 announces
  - No guarantees
- After *n* more nodes appended
  - Everyone sees that thread *k*+1 wants help
  - Everyone tries to append that node
  - Someone succeeds

Then

- After thread A announces its node
- No more than *n* other calls
  - Can start and finish
  - Without appending A's node

wait-free

```
while (announce[i].seq == 0) {
  Node before = head[i];
   Node help = announce[(before.seq + 1) % n];
   if (help.seq == 0)
       prefer = help;
   else
       prefer = announce[i];
  Node after = before.decideNext.decide(prefer);
  before.next = after;
  after.seq = before.seq + 1;
  head[i] = after;
 }
```

same as before

Compute my response

```
...
//compute my response
SeqObject MyObject = new SeqObject();
current = tail.next;
while (current != announce[i]){
    MyObject.apply(current.invoc);
    current = current.next;
   }
return MyObject.apply(current.invoc);
}
```

- Compute result by
  - sequentially applying list's method calls
  - to a private copy of the object
  - starting from the initial state

k-set agreement

Consensus generalized: k-Set Agreement

A process *proposes* an *input* value in V (|V|≥k+1) and tries to *decide* on an *output* value in V

- *k-Agreement:* At most k distinct values are decided
- *Validity:* Every decided value is a proposed value
- *Termination:* No process takes infinitely many steps without deciding
  (Every correct process decides)

1-set agreement = consensus

(N-1)-set agreement = set agreement

Impossibility of Set Agreement

Theorem 2 No wait-free algorithm solves set agreement in read-write

- Starting with N values, there is no way to drop one (decide on at most N-1)

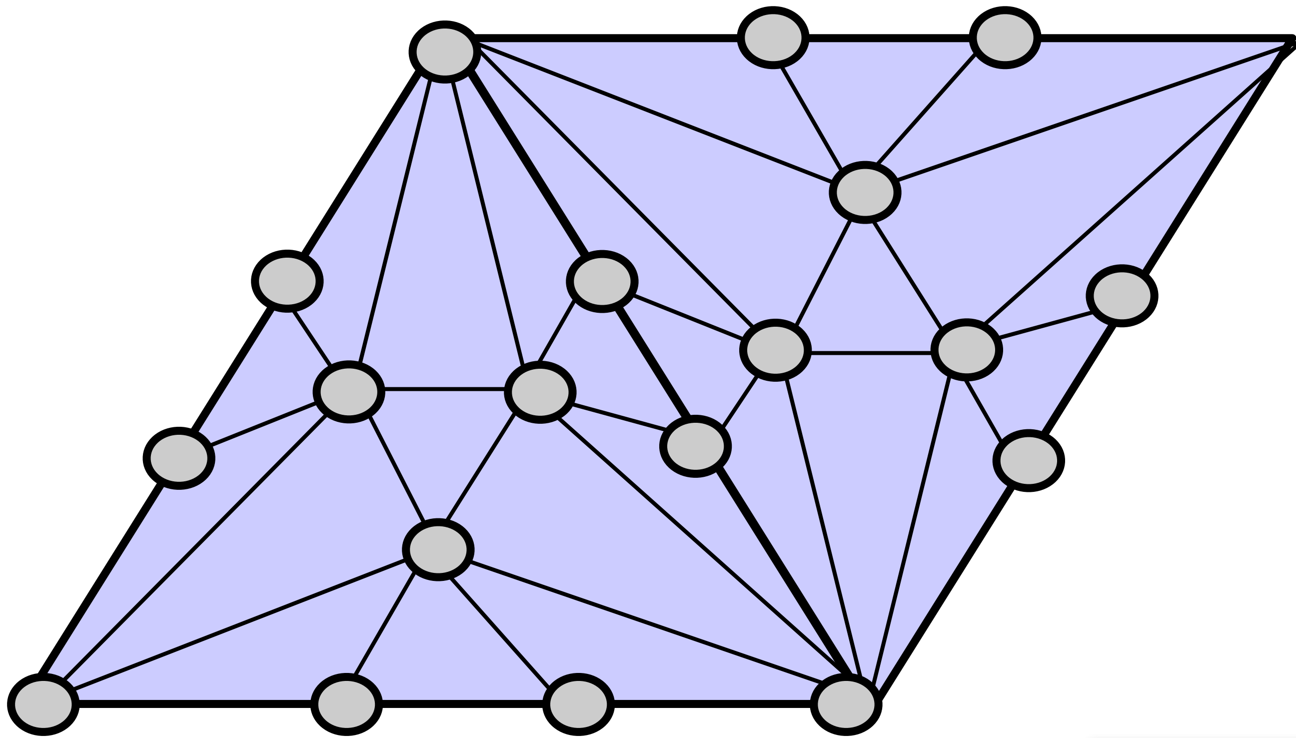
- Implies the impossibility of wait-free k-set agreement for all $k \leq N-1$

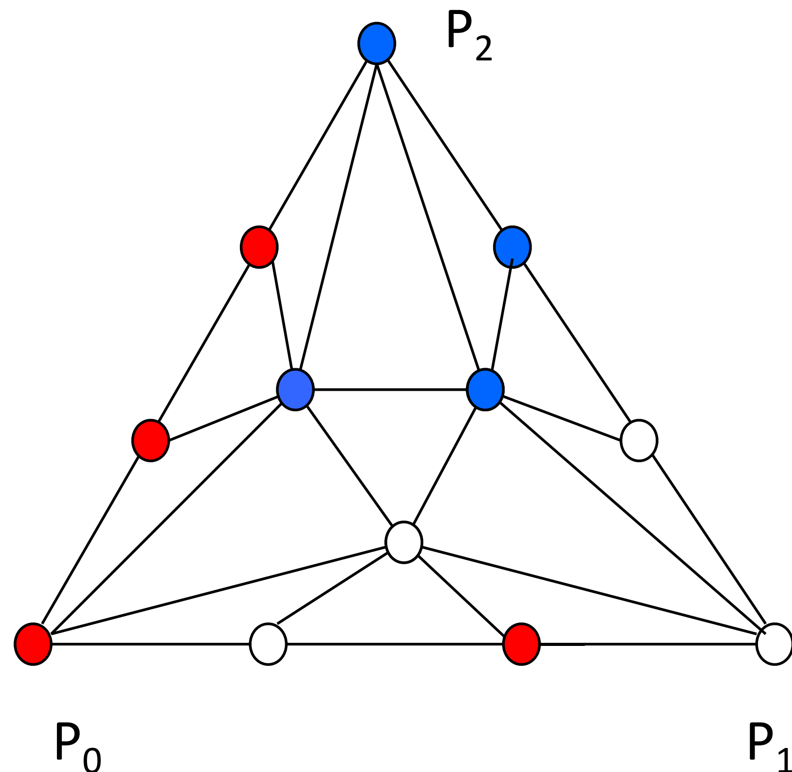
Solving Set Agreement in IIS

- Each p_i proposes its id i
- There is an IIS round r, such that each process (that reaches round r) outputs some id
- In each run, the set of output ids is a subset of participants of size ≤N-1

Implies Sperner coloring of the subdivided simplex IIS^r

- B is a subdivision of A

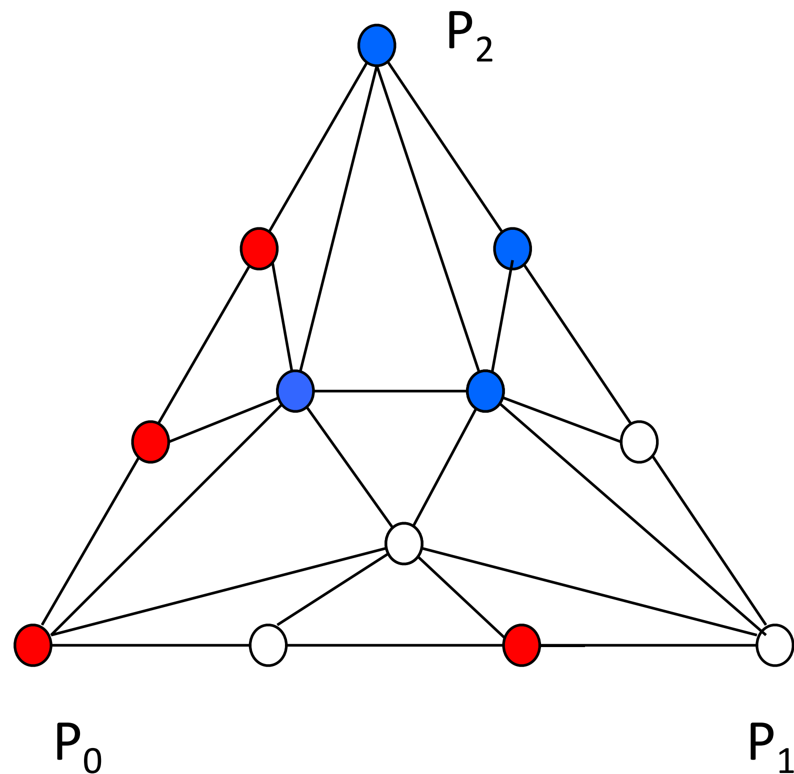




For each simplex s of B there is a simplex t of A such that $|s| \subseteq |t|$.

For each simplex s of A, $|s|$ is the union of a finite set of geometric simplexes of B.

Sperner Coloring



"Corners" have distinct colors

Edge vertexes have corner colors

Every vertex has face boundary colors

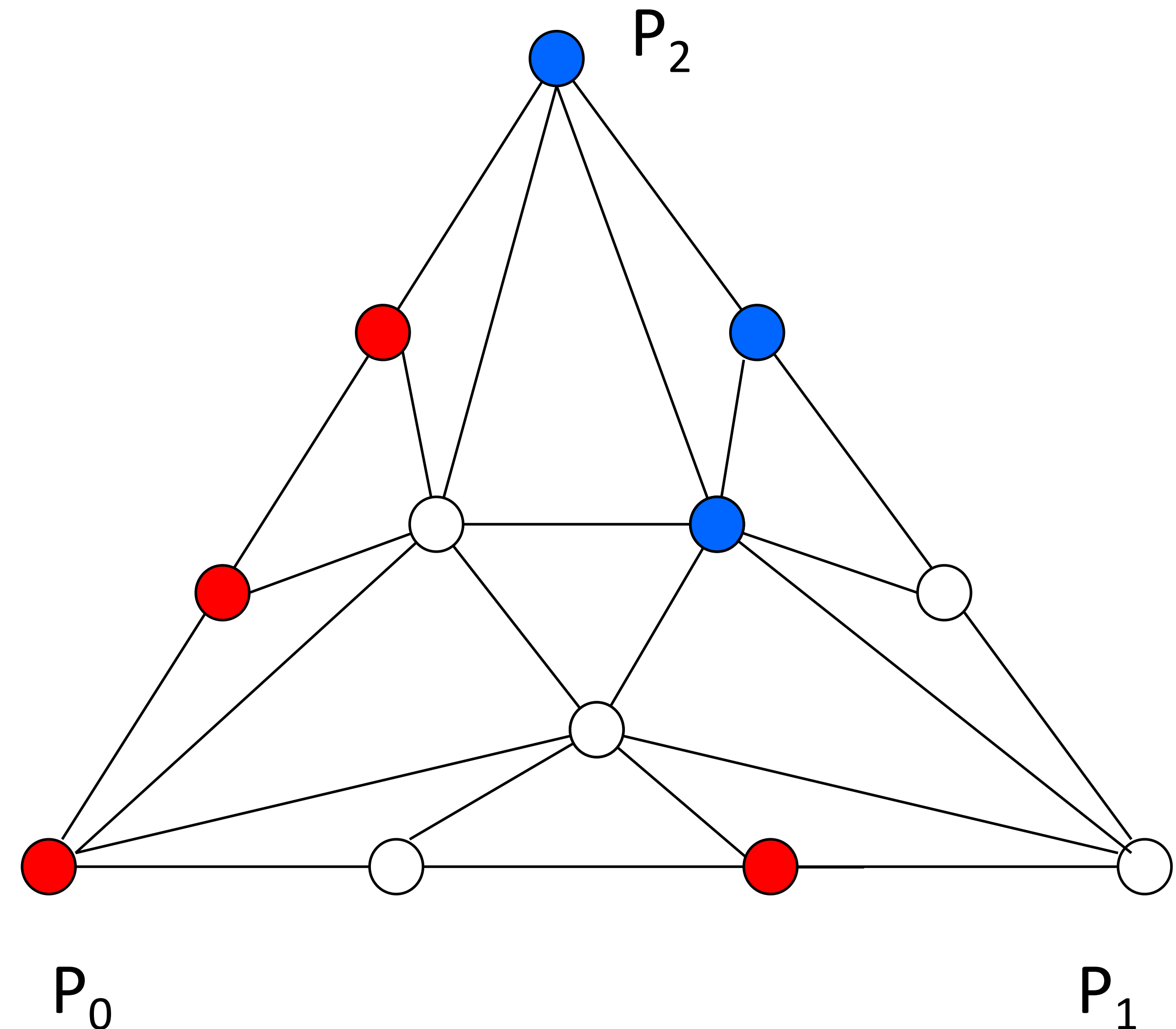
Sperner Lemma: In any Sperner coloring, at least one *n*-simplex has all *n+1* colors

Assume there is a wait-free protocol solving set agreement

after r rounds IIS we get a subdivision of S^{n-1} each node is coloured by its decision, we get a Sperner coloring => N different values are decided

Sperner's Lemma

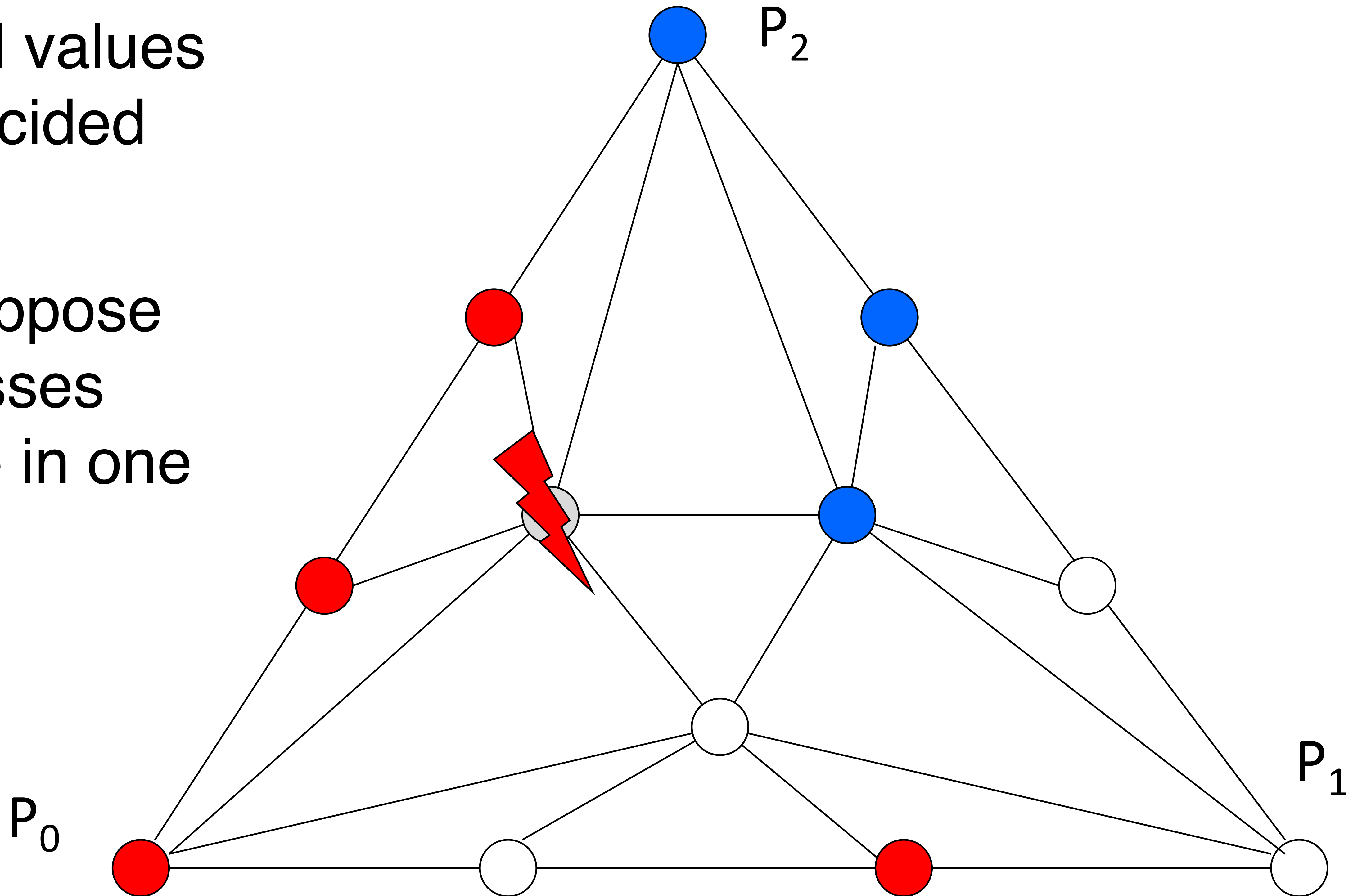
Sperner's coloring of any subdivision D of S^{N-1} has a simplex with all N colors



Corollary: wait-free set agreement is impossible in IIS (and, thus, in read-write)

In at least one IIS run, N values are decided

Here: suppose processes decide in one round

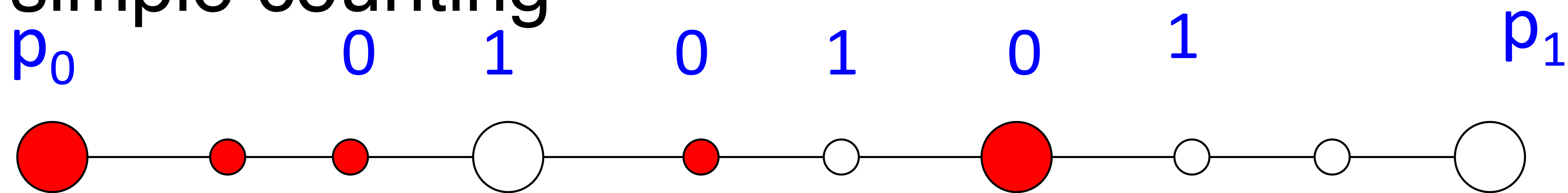


Sperner's lemma: inductive step

Claim: for each k=0,…,N-1, face $p_0,\ldots,p_k$ of S^{N-1} has an odd number of k-dimensional simplexes colored 0,…,k (odd then at least one!)

By induction: k=0 - trivial (exactly one)

k=1, simple counting



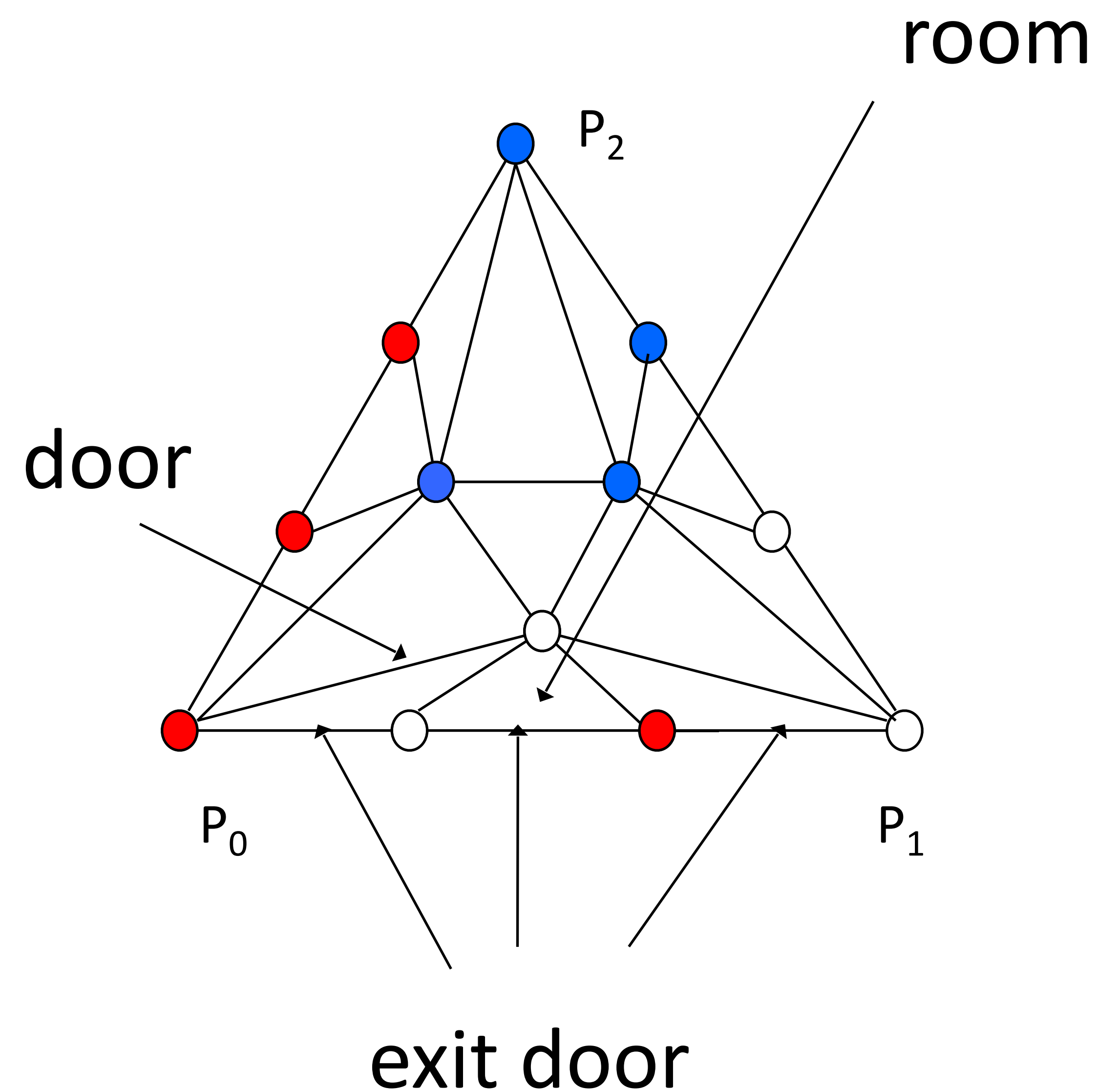
Suppose the claim holds for k<N-1 and consider the face 0,…,k

Sperner's: rooms and doors

Each k-simplex contained in face $p_0,\ldots,p_k$ is a room

A (k-1)-dimensional face (a subset of k-1 vertices) of a room colored in $0,\ldots,k-1$ is a door

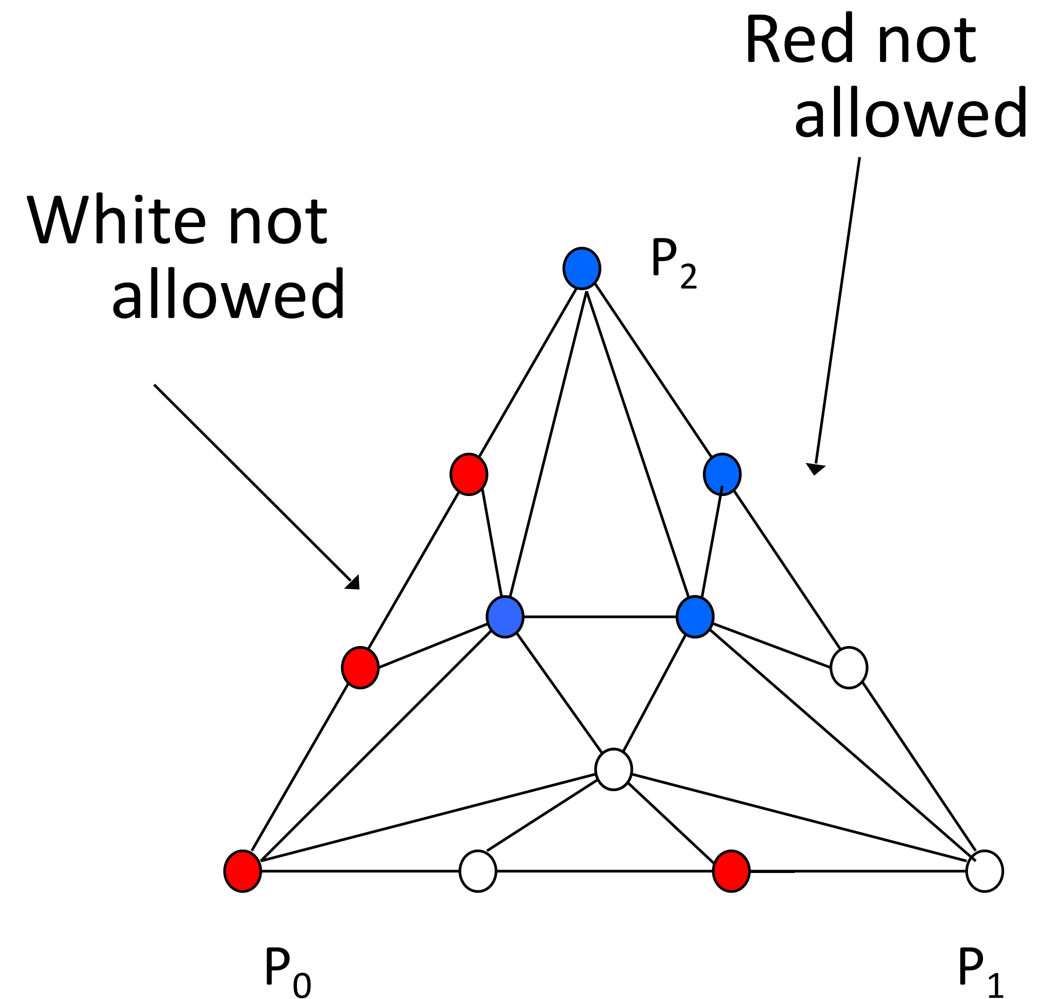
A door is an exit if it is contained in the boundary of $p_0,\ldots,p_k$



Sperner's: rooms and doors

There is an odd number of exits!

- By Sperner's coloring no faces other than in $p_0,\ldots,p_{k-1}$ can contain simplexes colored 0, …,k-1
- Exits may only be contained in $p_0,\ldots,p_{k-1}$
- By induction, $p_0,\ldots,p_{k-1}$ contains an odd number of doors (colored with 0,…,k-1)



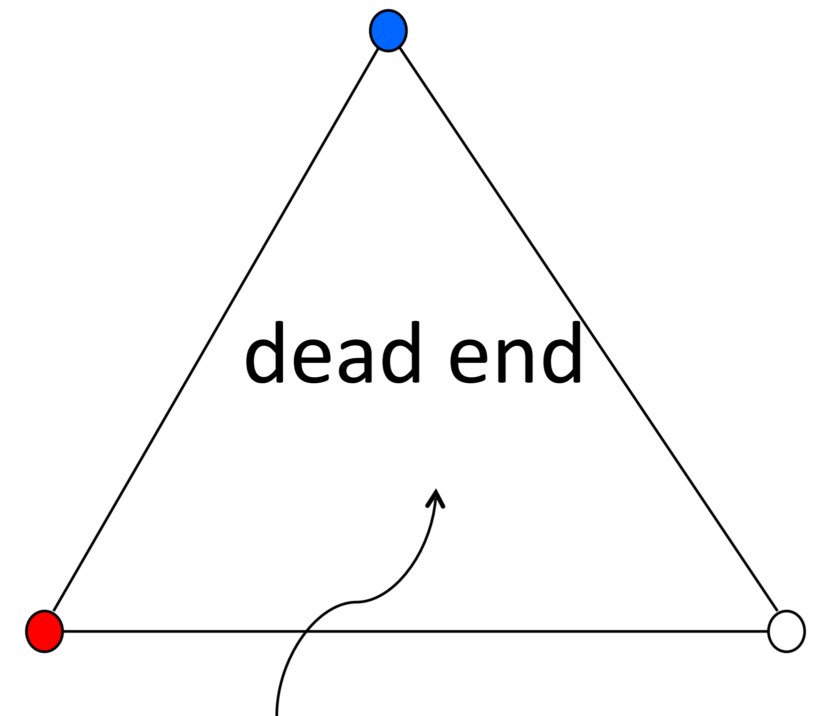
Sperner's: corridors and dead ends

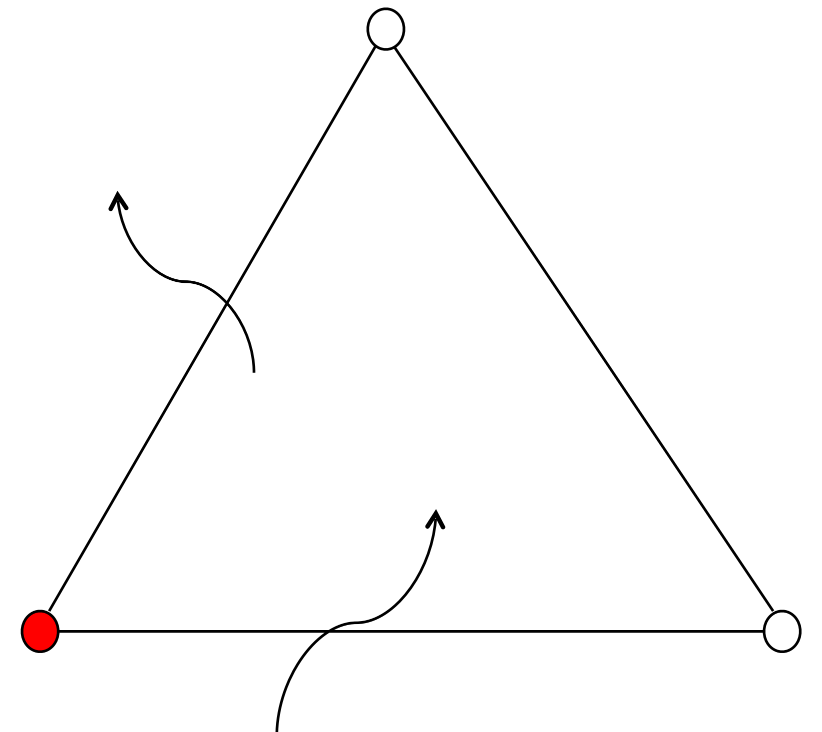
Consider a room with a door (k vertices colored 0,…,k-1)

Two cases possible (depending on the color of the remaining vertex):

- the remaining vertex is colored k - the room has exactly *one* door (dead end)
  - ✓ the room is fully colored (0,…,k)
- The remaining vertex is colored in one of {0,…,k-1} - the room has *two* doors

We show that there is an odd number of dead ends contained in face $p_0,\ldots,p_k$



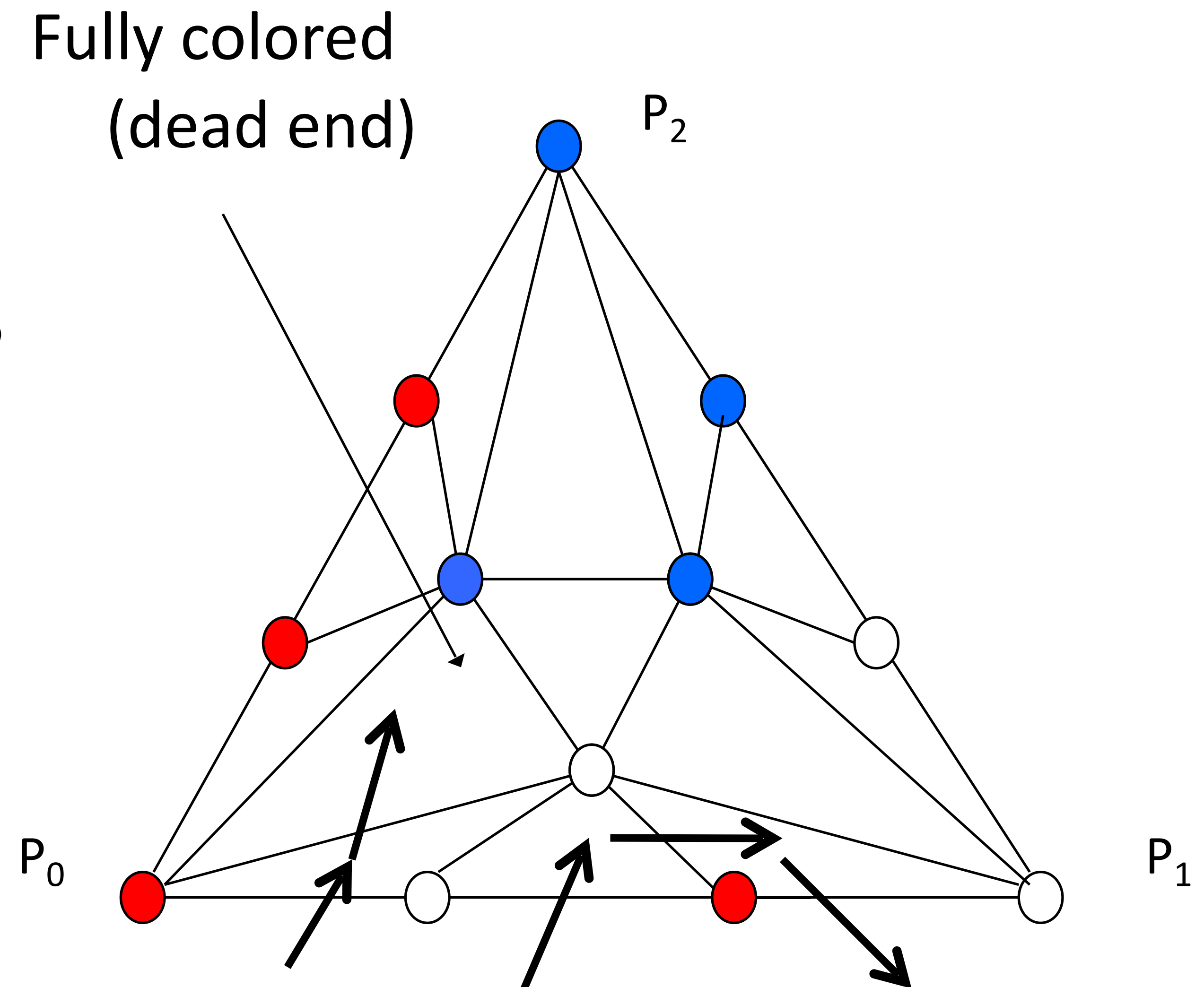


Sperner's: counting fully colored rooms

Start with an exit and walk through the doors

There are only finitely many rooms, thus, two only cases possible:

- Stop in a dead end (fully colored simplex)
- Reach another exit (two exits for such a walk)



Sperner's: crossing the rooms

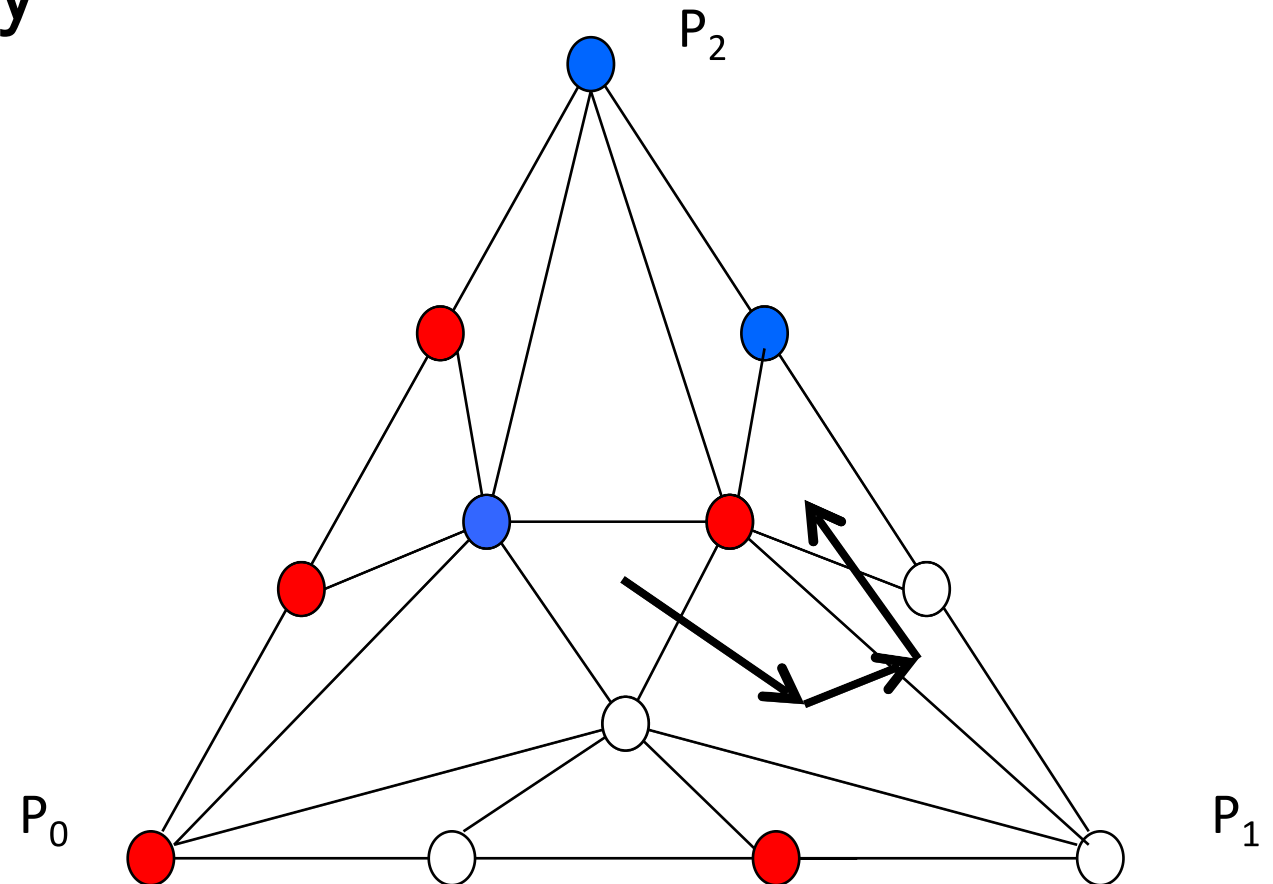
The number of exits is odd

=> The number of dead ends (fully colored simplexes) reachable from exit is odd!

Now start in a dead end not reachable from an exit: can only stop in another dead end not reachable from an exit

⇒ The number of dead ends not reachable from exit is even!

⇒ The total number of fully colored rooms is odd



Thus

- No algorithm can wait-free solve (N-1)-set agreement in IIS
  - ✓ otherwise Sperner's coloring of some IIS^r would have no fully colored simplexes (but there is at least one!)
- No algorithm can wait-free solve set agreement in AS (or any read-write model):
  - ✓ otherwise we can (non-blocking) simulate it in IIS and thus find a wait-free algorithm in IIS
- We cannot tolerate N-1 failures: can we tolerate less?
  - ✓ E.g., can we solve consensus (1-set agreement) 1-resiliently?

So…

- No algorithm can wait-free (N-resiliently) solve consensus

- We cannot tolerate N-1 failures: can we tolerate less?
  - ✓E.g., can we solve consensus 1-resiliently?

1-resilient consensus?

What if we have 1000000 processes and one of them can crash?

NO

We present a direct proof now

(an indirect proof by reduction to the wait-free impossibility also exists)

Impossibility of 1-resilient consensus [FLP85,LA87]

Theorem 2 No 1-resilient (assuming that one process might fail) algorithm solves consensus in read-write

Proof

By contradiction, suppose that an algorithm A solves 1-resilient binary consensus among $p_0,\ldots p_{N-1}$

Proof

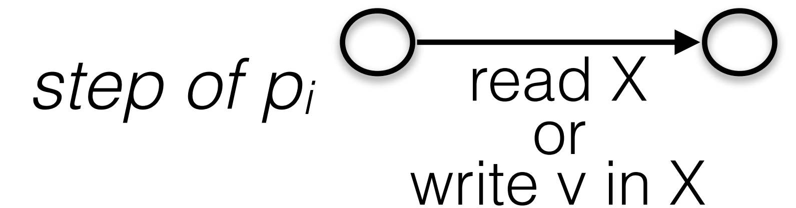
A *configuration (or state)* = state of each process + state of shared memory

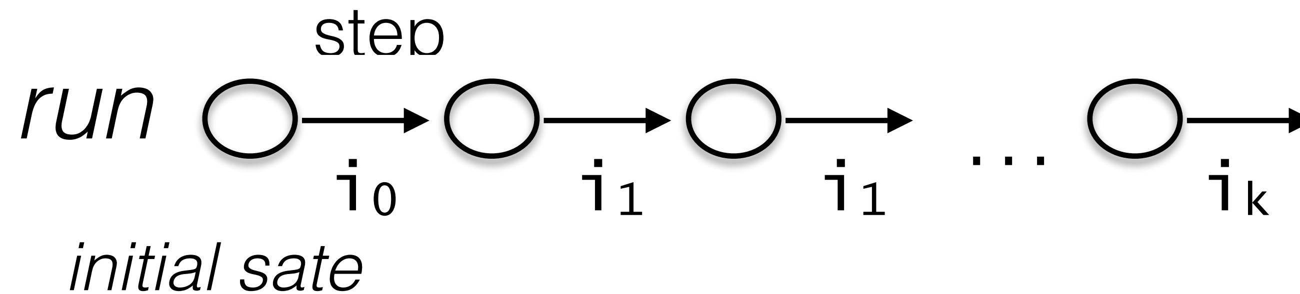
Initial configuration = one input value per process + initial state of processes and shared memory

A *step* = atomic step (read or write on shared memory) of a process

run of A is a sequence of step applied to the initial state

A *schedule* is a sequence of process ids $i_1, i_2, \ldots i_k, \ldots$





A run of A can be seen as and initial input configuration (one input per process) and a schedule

Every correct (taking sufficiently many steps) process decides!

Proof: valence

Let R be a finite run

- We say that R is *v-valent* (for v in {0,1}) if v is decided in every infinite extension of R

- We say that R is *bivalent* if R is neither 0-valent nor 1-valent
  (there exists a 0-valent extension of R and a 1-valent extension of R)

- We say R is *univalent* if R is not bivalent

Proof: valence claims

Claim 1 Every finite run is 0-valent, or 1-valent, or bivalent.
(by Termination)

Claim 2 Any run in which some process decides v is v-valent
(by Agreement)

Corollary 1: No process can decide in a bivalent run (by Agreement).

Bivalent input

Claim 3 There exists a bivalent input configuration (empty run)

Proof

Suppose not: Consider sequence of input configurations C_0, …,C_N:

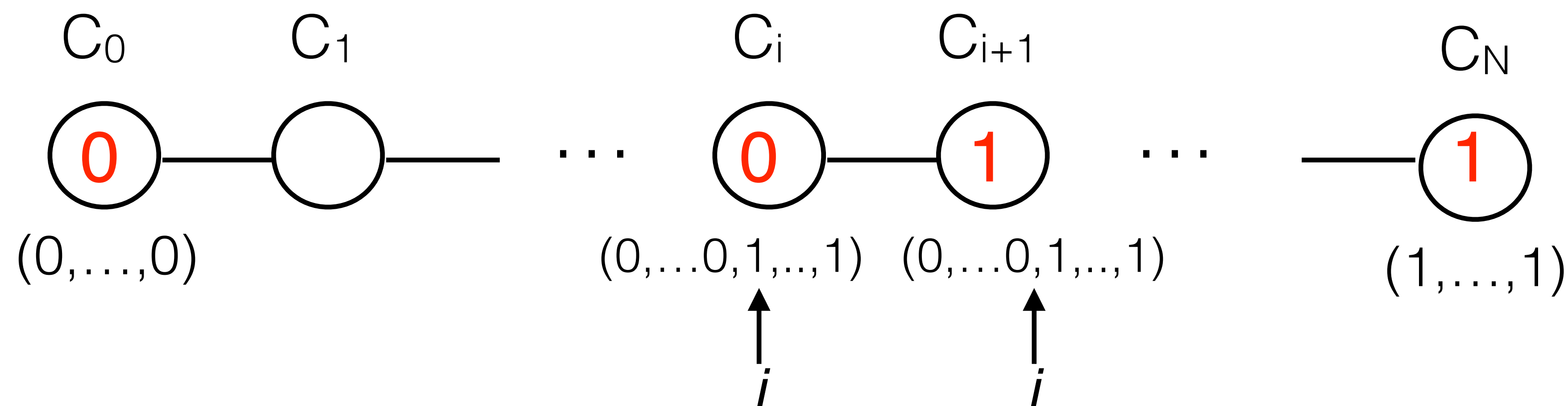
in C_i: p_0,..,p_{i-1} propose 1, and p_i,..,p_N propose 0

- All C_i's are univalent (hypothesis)
- C_0 is 0-valent (by Validity)
- C_N is 1-valent (by Validity)

Bivalent input

There exists i in {0,…N-1} such that C_i is 0-valent and C_{i+1} is 1-valent!

C_i and C_{i+1} differ only in the input value of p_i (it proposes 1 in C_i and 0 in C_{i+1})



Consider a run R starting from C_i in which p_i takes no steps (crashes initially): eventually all other processes decide 0

Consider R' that is like R except that it starts from C_{i+1},

R and R' are indistinguishable!

- Thus, every process decides 0 in R' --- contradiction (C_{i+1} is 1-valent)

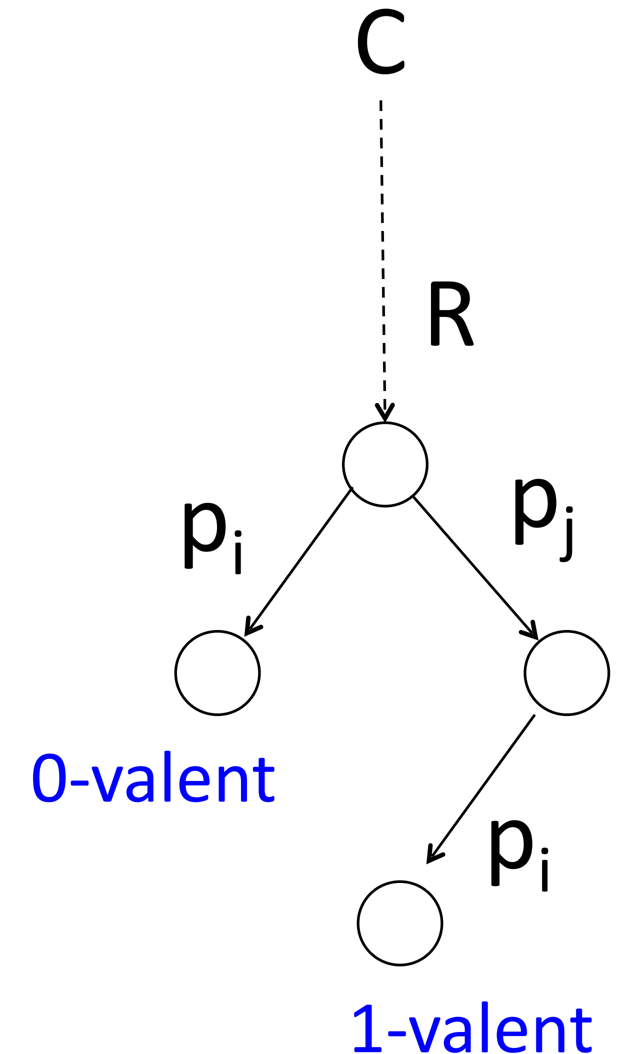
Critical run

Claim 4 There exists a finite run R and two processes p_i and p_j such that R.i is 0-valent and R.j.i is 1-valent (or vice versa)

(R is called critical)

Proof of Claim 4: By construction, take the bivalent empty run C (by Claim 3 it exists)

We construct an ever-extending fair (giving each process enough steps) critical run



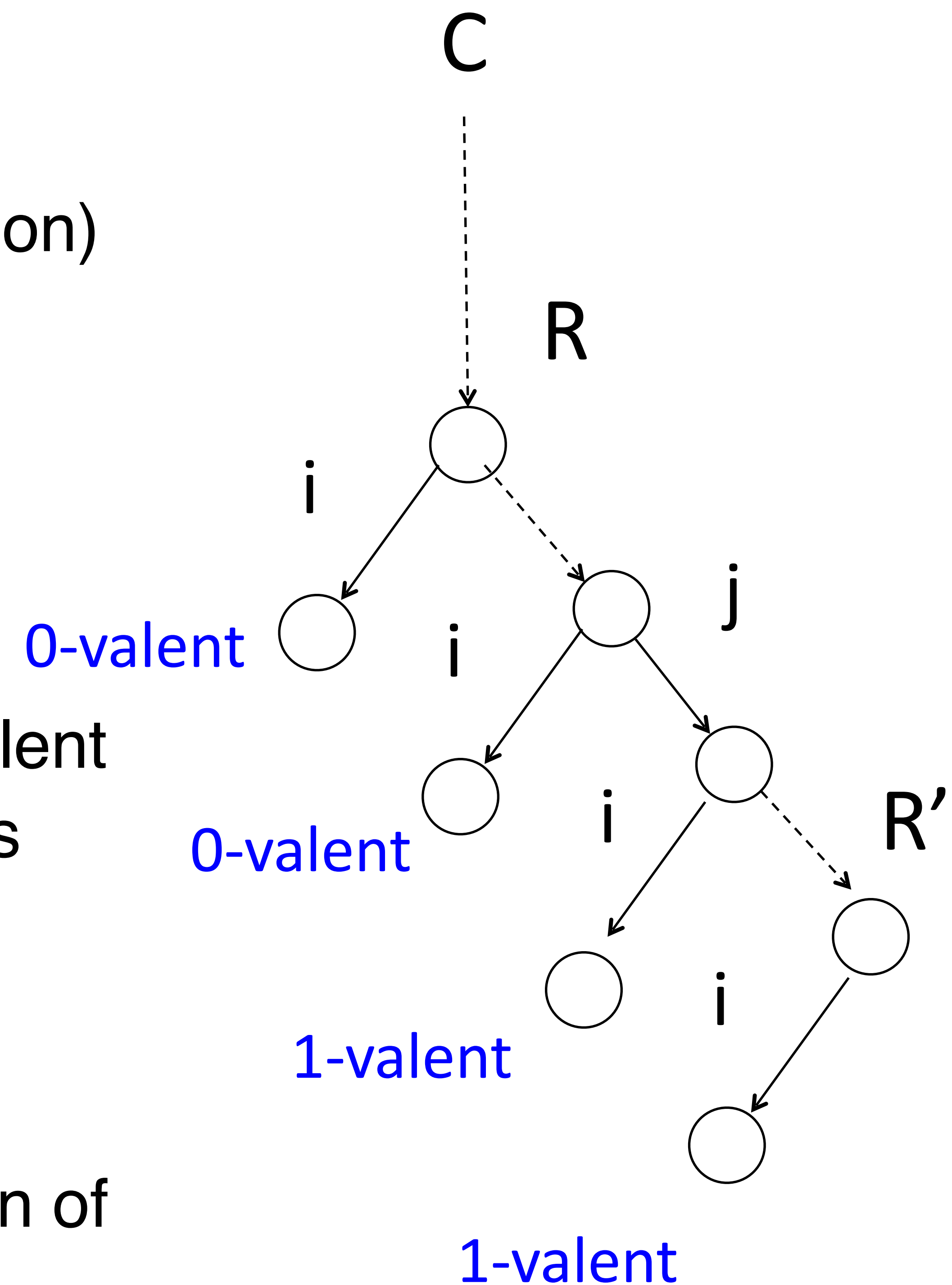
Proof of Claim 4: critical run

repeat forever

take the next process p_i (in round-robin fashion)

if for some R', an extension of R, R'.i is bivalent **then** R:=R'.i

else stop



- If never stops – ever extending (infinite) bivalent runs in which every process is correct (takes infinitely many steps) – contradiction with termination
- If stops – (suppose R.i is 0-valent) – R is bivalent then consider a 1-valent R' extension of R

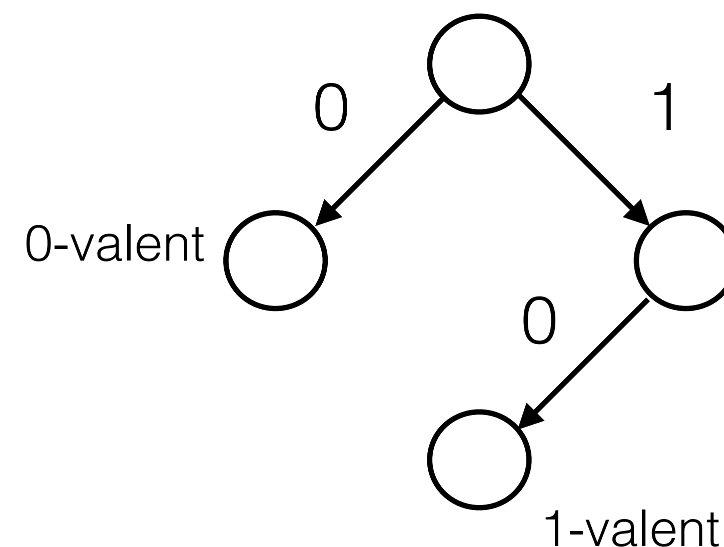
✓ There is a critical configuration between R and R'

Proof (contd.): the next steps in R

(Let p_0=i and p_1=j)

Four cases, depending on the next steps of p_0 and p_1 in R

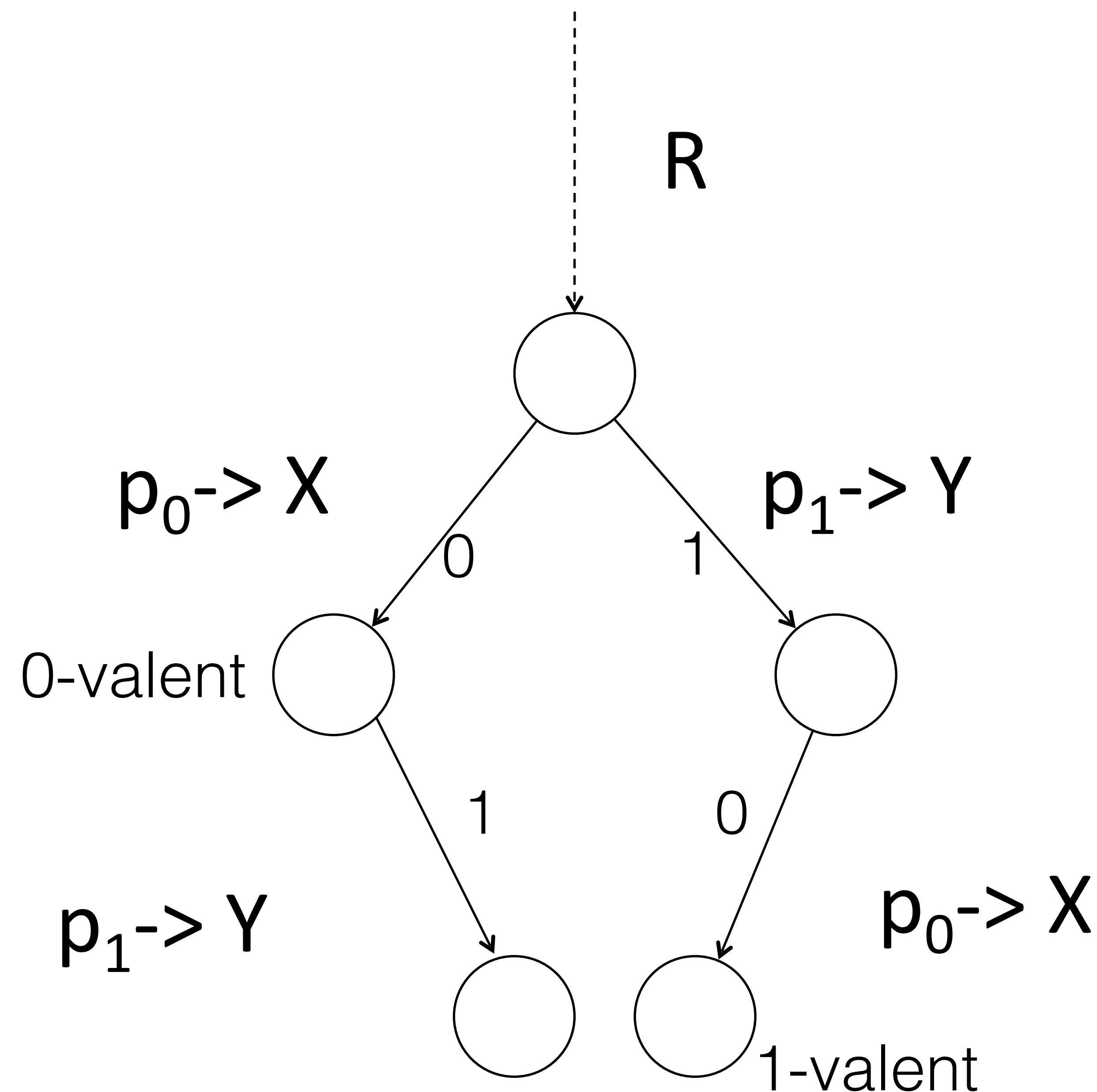
- p_0 and p_1 are about to access different objects (registers) in R
- p_1 reads X and p_0 reads X
- p_0 writes in X
- p_1 reads X



Proof (contd.): cases and contradiction

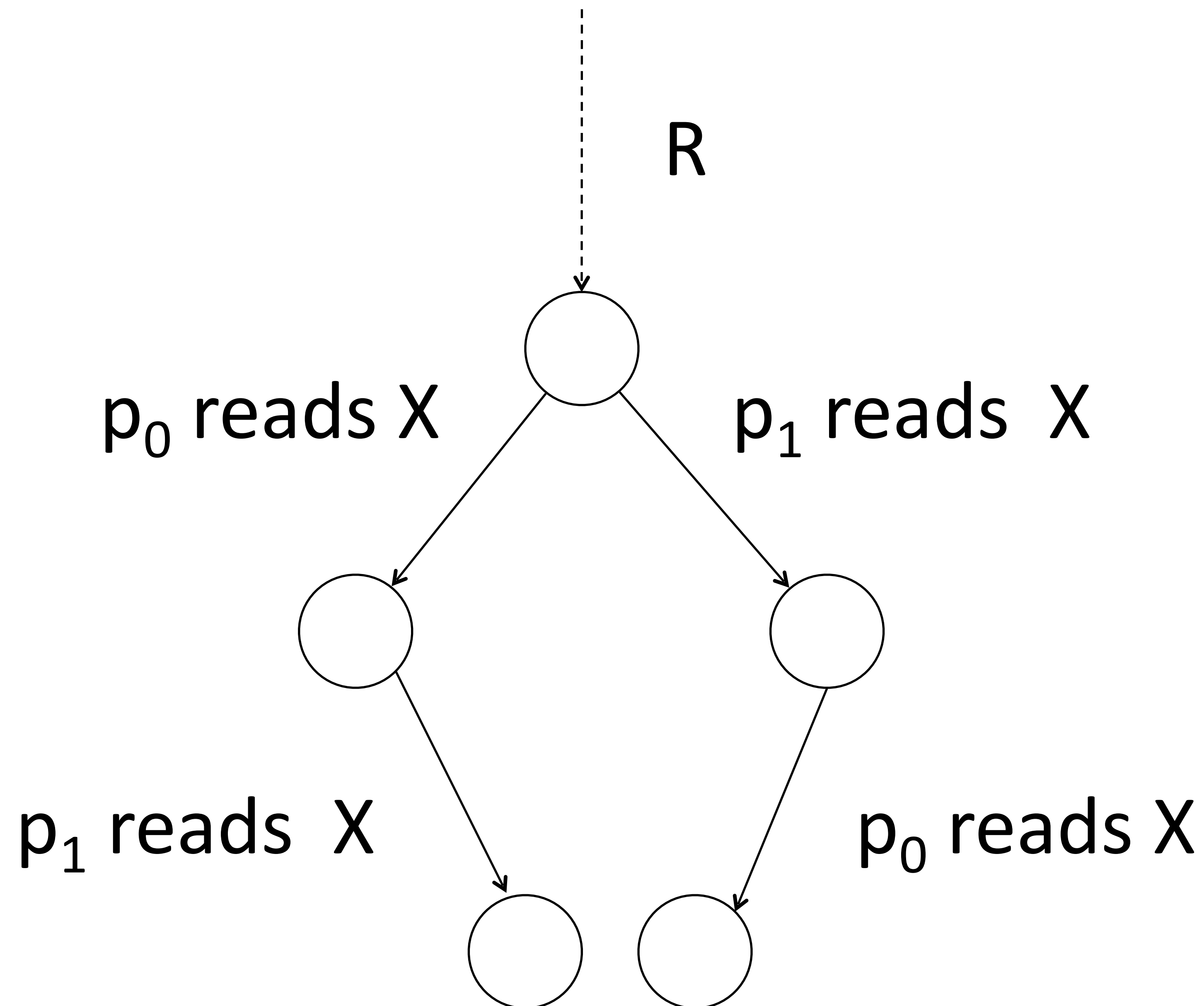
- p_0 and p_1 are about to access different objects in R

✓R.0.1 and R.1.0 are indistinguishable



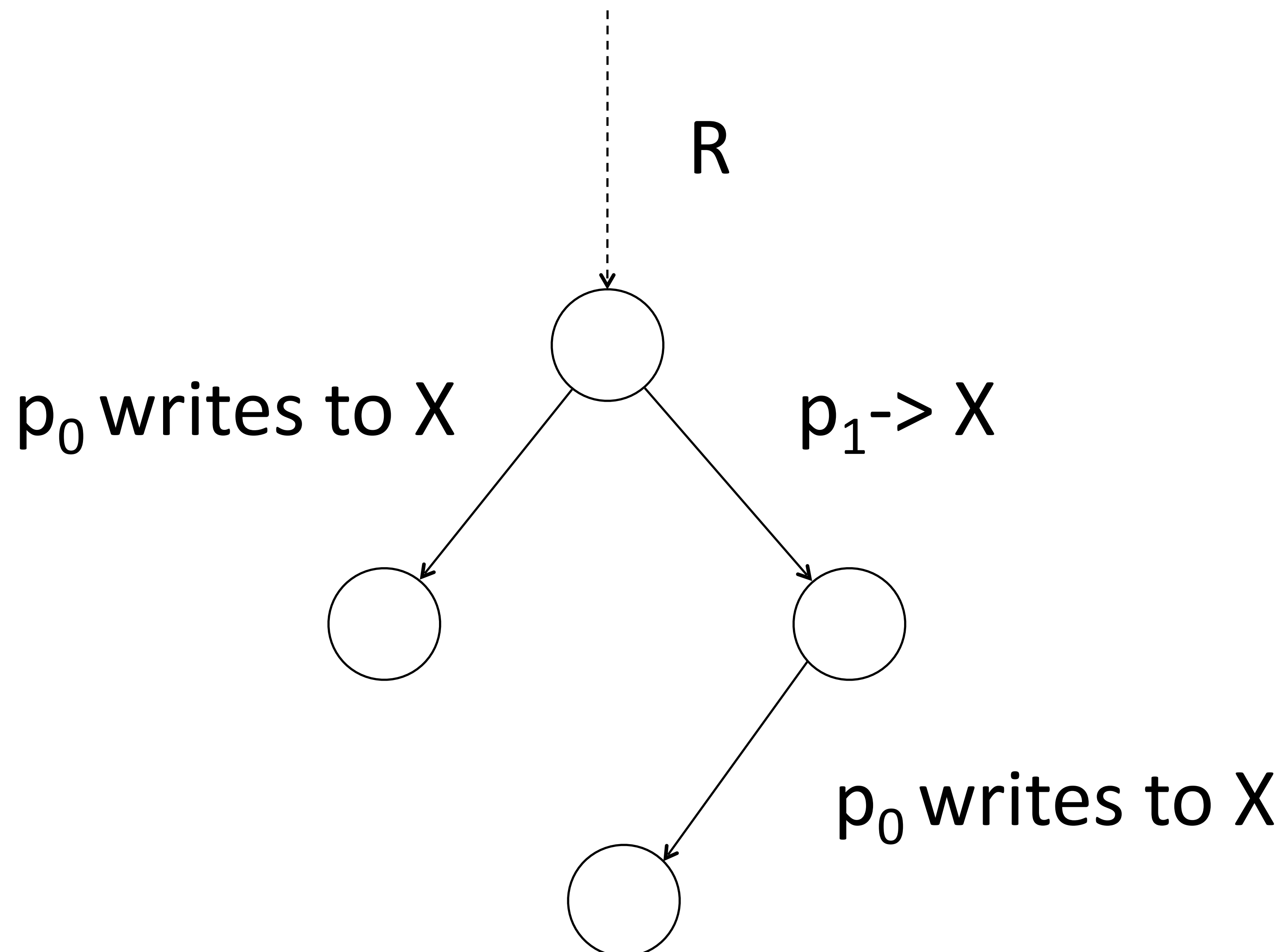
Proof (contd.): cases and contradiction

- p_0 and p_1 are about to read the same object X
  R.0.1 and R.1.0 are indistinguishable



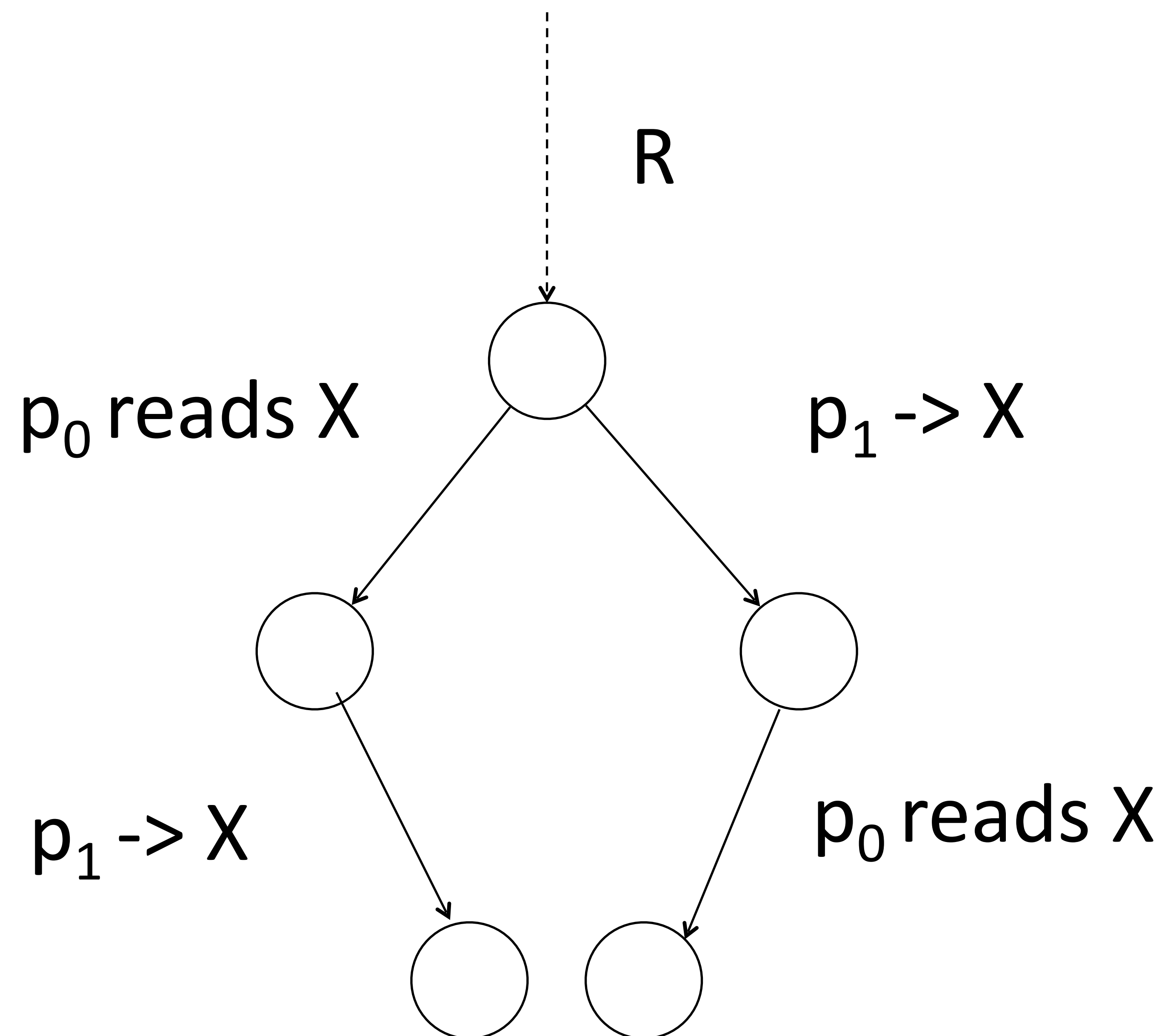
Proof (contd.): cases and contradiction

- p_0 is about to write to X
  - ✓Extensions of R.0 and R.1.0 are indistinguishable for all except p_1 (assuming p_1 takes no more steps)



Proof (contd.): cases and contradiction

- p_0 is about to read to X
  ✓Extensions of R.0.1 and R.1.0 are indistinguishable for all but p_0 (assuming p_0 takes no more steps)



Thus

- No critical run exists
- A contradiction with **Claim 4**

⇒ 1-resilient consensus is impossible in read-write