# Computer Science 425 Distributed Systems (Fall 2009)

Lecture 10

The Consensus Problem

Part of Section 12.5 and

Paper: “Impossibility of Distributed Consensus with One Faulty Process” Fisher, Lynch, Paterson, JACM, 1985 (Sections 1-3)

# Acknowledgement


- The slides during this semester are based on ideas and material from the following sources:
  - Slides prepared by Professors M. Harandi, J. Hou, I. Gupta, N. Vaidya, Y-Ch. Hu, S. Mitra.
  - Slides from Professor S. Gosh's course at University o Iowa.

# Administrative

- MP1 posted September 8, Tuesday
  - Deadline, September 25 (Friday), 4-6pm Demonstrations
  - Readme Files Due on September 28 (Monday)
    - **Email** readme documentation of your MP1 to TA
- HW 2 posted September 22, Tuesday
  - Deadline, October 6 (Tuesday), 2pm (at the beginning of the class)
- **HW1 grading scale and histogram posted**

# Give it a thought

Have you ever wondered why vendors of (distributed) software solutions always only offer solutions that promise five-9's reliability, seven-9's reliability, but never 100% reliability?

# Give it a thought

Have you ever wondered why software vendors always only offer solutions that promise five-9's reliability, seven-9's reliability, but never 100% reliability?

The fault does not lie with Microsoft Corp. or Apple Inc. or Cisco

The fault lies in the ***impossibility of consensus***

# What is Consensus?

- $N$ processes
- Each process $p$ has
  - input variable $x_p$ ($v$) : initially either **0** or **1**
  - output variable $y_p$ ($d$) : initially **b** (**b**=undecided)
    - $v$ – single value for process $p$; $d$ – decision value
- A process **is non-faulty** in a run provided that it takes infinitely many steps, and it is faulty otherwise
- **Consensus problem**: design a protocol so that either
  1. all non-faulty processes set their output variables to **0**
  2. all non-faulty processes set their output variables to **1**
  3. There is at least one initial state that leads to each outcomes 1 and 2 above

# Canonical Application

- A set of servers implement a distributed database
  - Subset of servers participate in a particular transaction
  - Some of the servers may fail
  - Remaining servers must **agree** on whether to install the results of the transaction to the database or discard them

# Solve Consensus!

- Uh, what's the **model**? (assumptions!)
- Assumptions:
  - Processes fail only by ***crash-stop***ping
  - Delivery channel is reliable
- **Synchronous system**: bounds on
  - Message delays
  - Max time for each process step

  e.g., multiprocessor (common clock across processors)
- Asynchronous system: no such bounds!

  e.g., The Internet! The Web!

# Consensus in Synchronous Systems (Dolev&Strong )

- For a system with at most *f* processes crashing, the algorithm proceeds in *f+1* rounds (with timeout), using basic multicast (B-multicast).
- $Values^r_i$: the set of proposed values known to process p=$P_i$ at the beginning of round *r*.
- Initially $Values^0_i = \{\}$ ; $Values^1_i = \{v_i = xp\}$

  for round r = 1 to f+1 do

    B-multicast (g, $Values^r_i$)

    $Values^{r+1}_i \leftarrow Values^r_i$

    on B-deliver($V_j$) from some process $p_j$

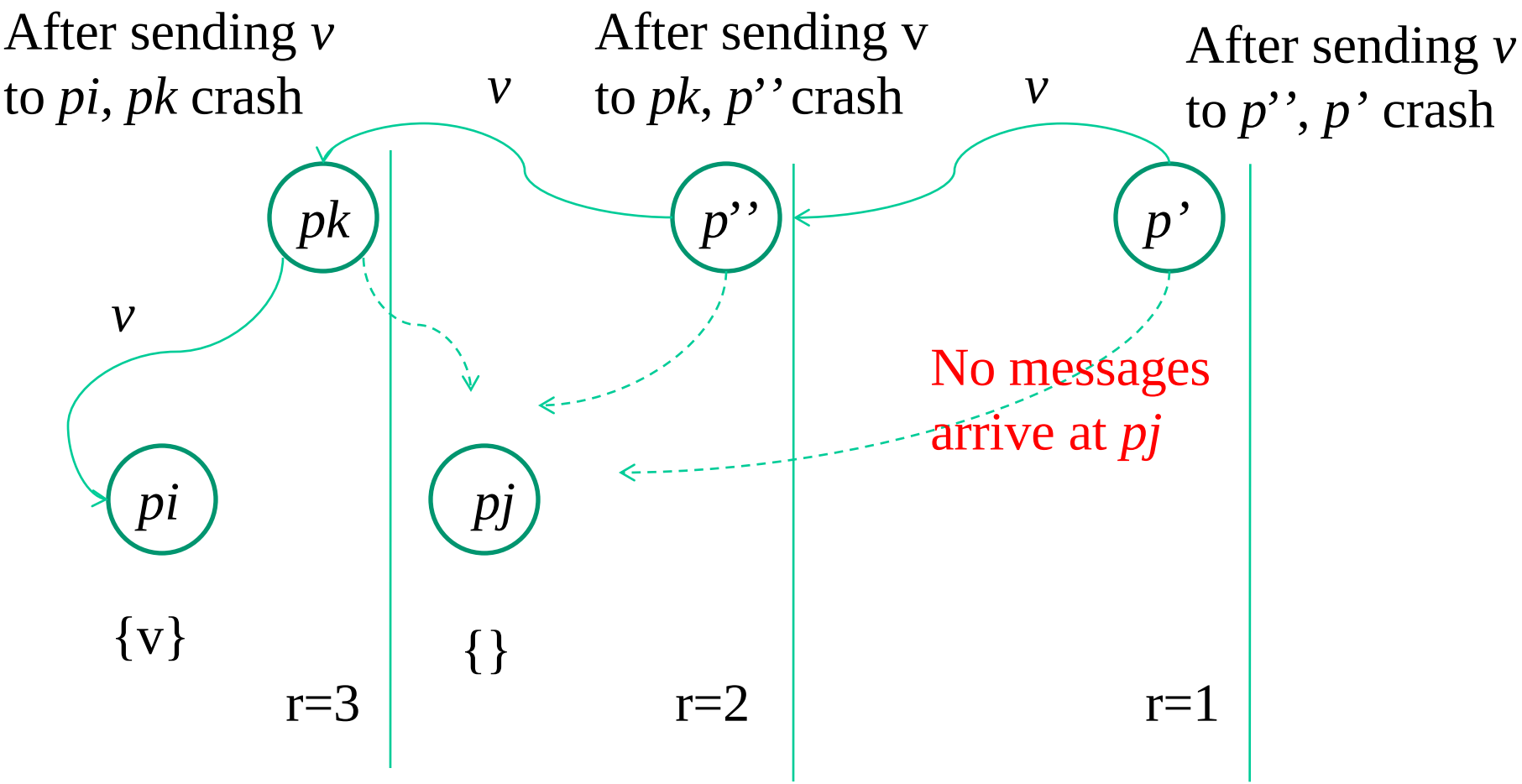
    $Values^{r+1}_i = Values^{r+1}_i \cup V_j$

    end

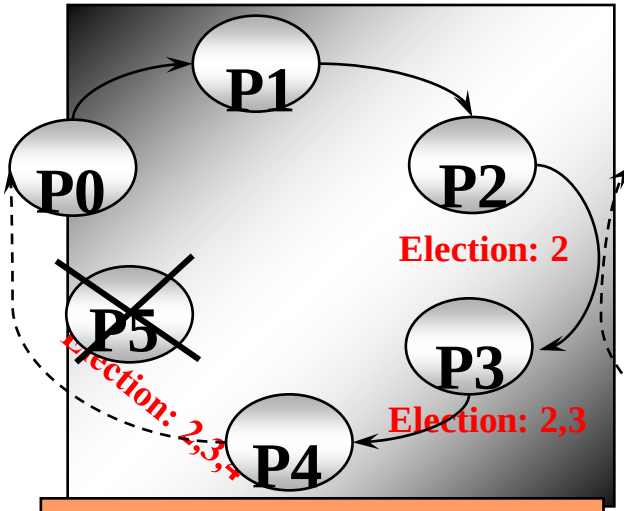
  end

**$yp=d_i$ = minimum($Values^{f+1}_i$)**

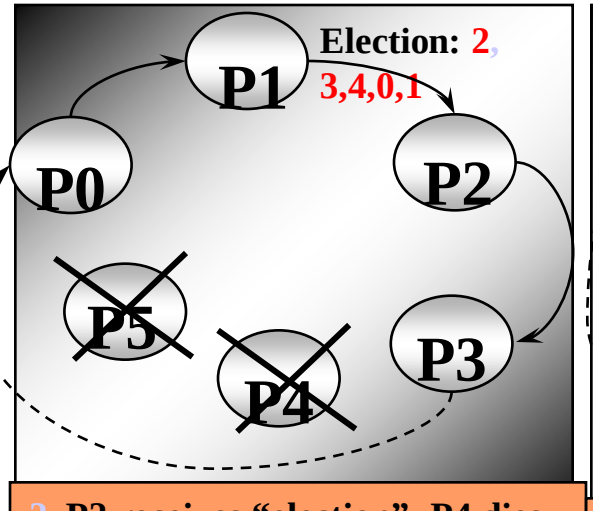
# Why does the algorithm work? (Proof by contradiction)



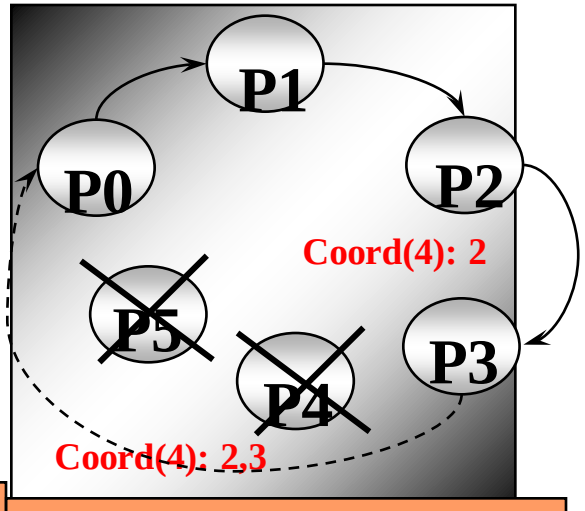
# Example of Consensus: Modified Ring Election for Synchronous Systems



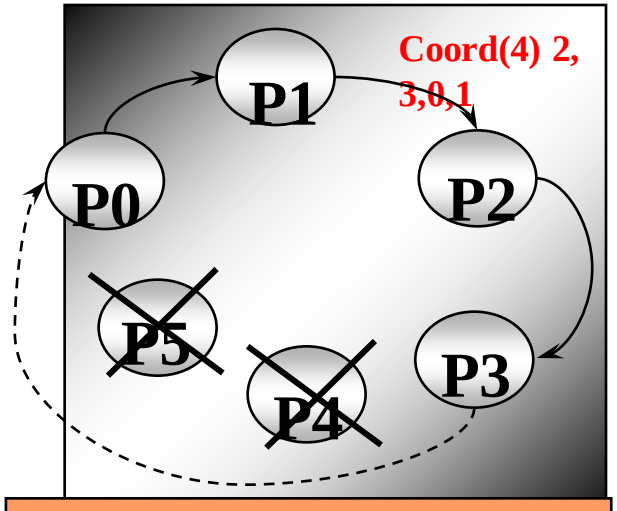

1. P2 initiates election



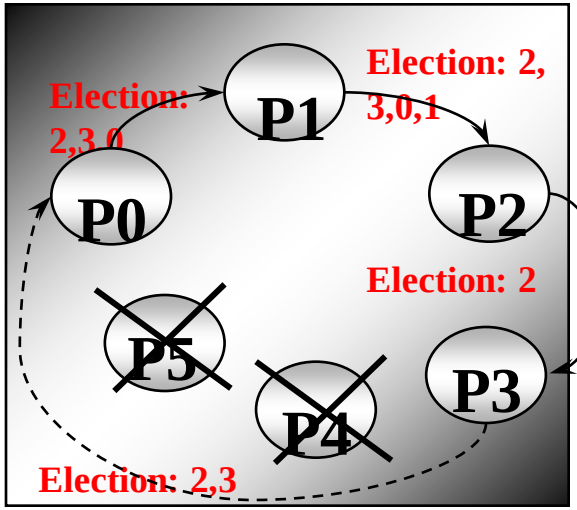

2. P2 receives "election", P4 dies



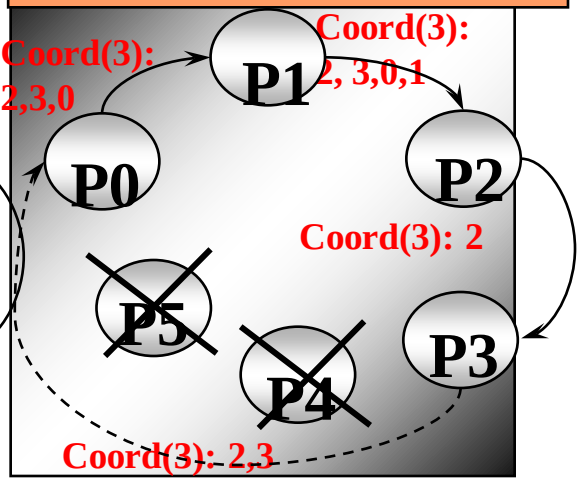

3. P2 selects 4 and announces the result




4. P2 receives "Coord", but P4 is not included




5. P2 re-initiates election




6. P3 is finally elected

# Consensus in Asynchronous Systems

# Consensus in an Asynchronous System

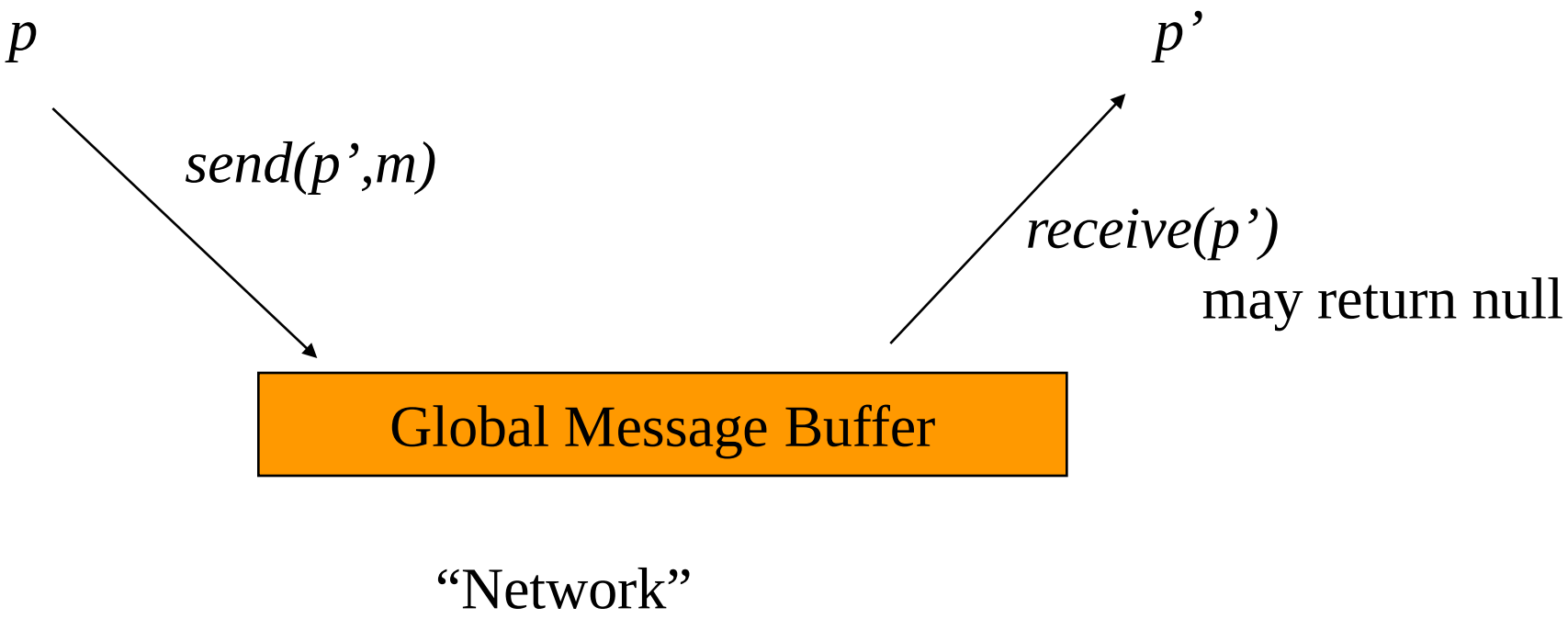
- Messages have arbitrary delay, processes arbitrarily slow (no timeouts!)
- Hence, Consensus is Impossible to achieve!
  - even a single failed process is enough to avoid the system from reaching agreement!
- Impossibility Applies to *any* protocol that claims to solve consensus!
- Proved in a now-famous result by **Fischer, Lynch and Patterson, 1983** (FLP)
  - Stopped many distributed system designers dead in their tracks
  - A lot of claims of "reliability" vanished overnight

# Recall

- Each process $p$ has an internal state
  - program counter, registers, stack, local variables
  - input register $x_p$ : initially either **0** or **1**
  - output register $y_p$ : initially **b** (**b**=undecided)
- **Consensus Problem**: design a protocol so that either
  1. all non-faulty processes set their output variables to **0**
  2. all non-faulty processes set their output variables to **1**
  3. (No trivial solutions allowed)

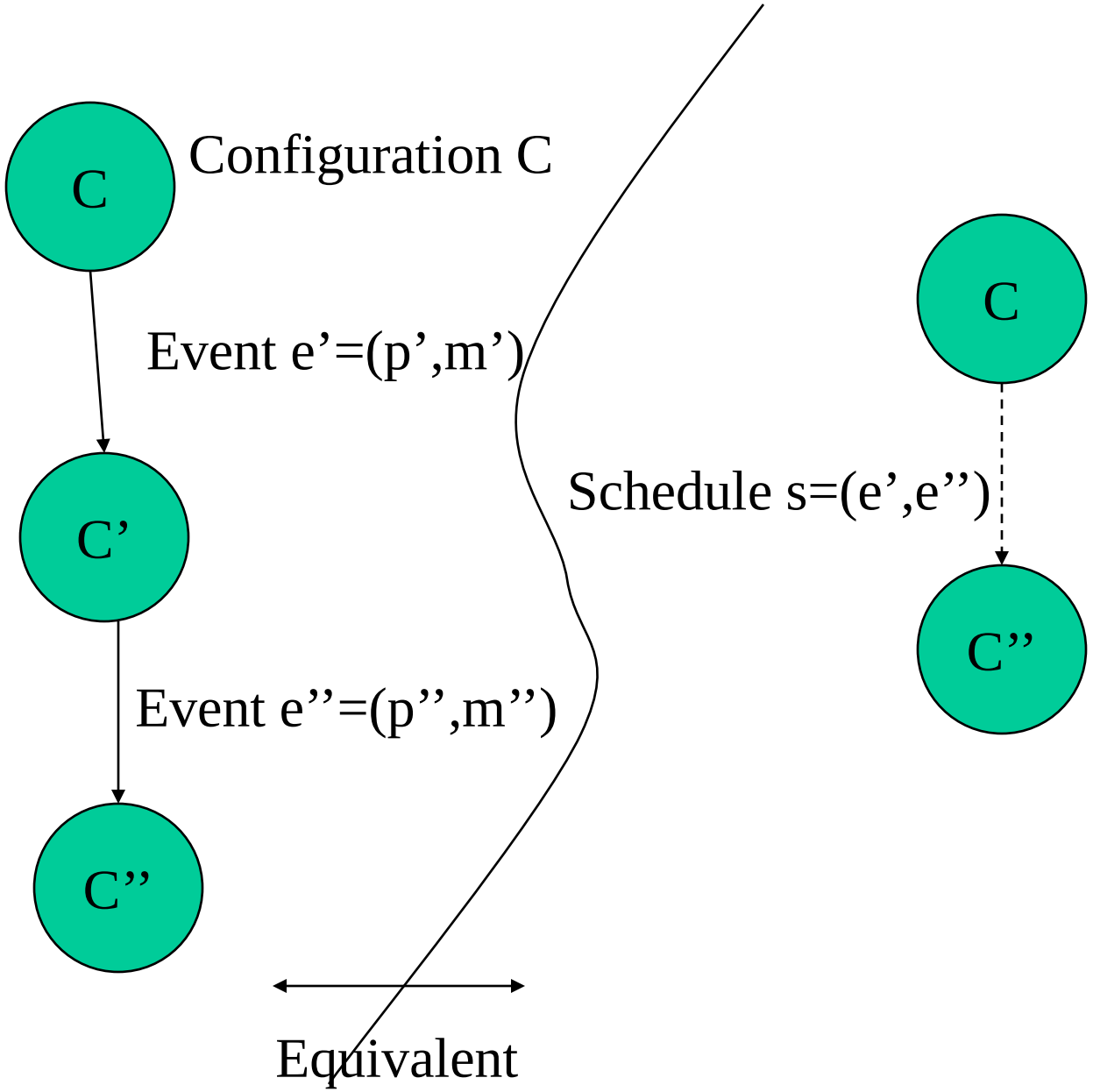
**Goal: Show Impossibility of Consensus!**

# Definitions



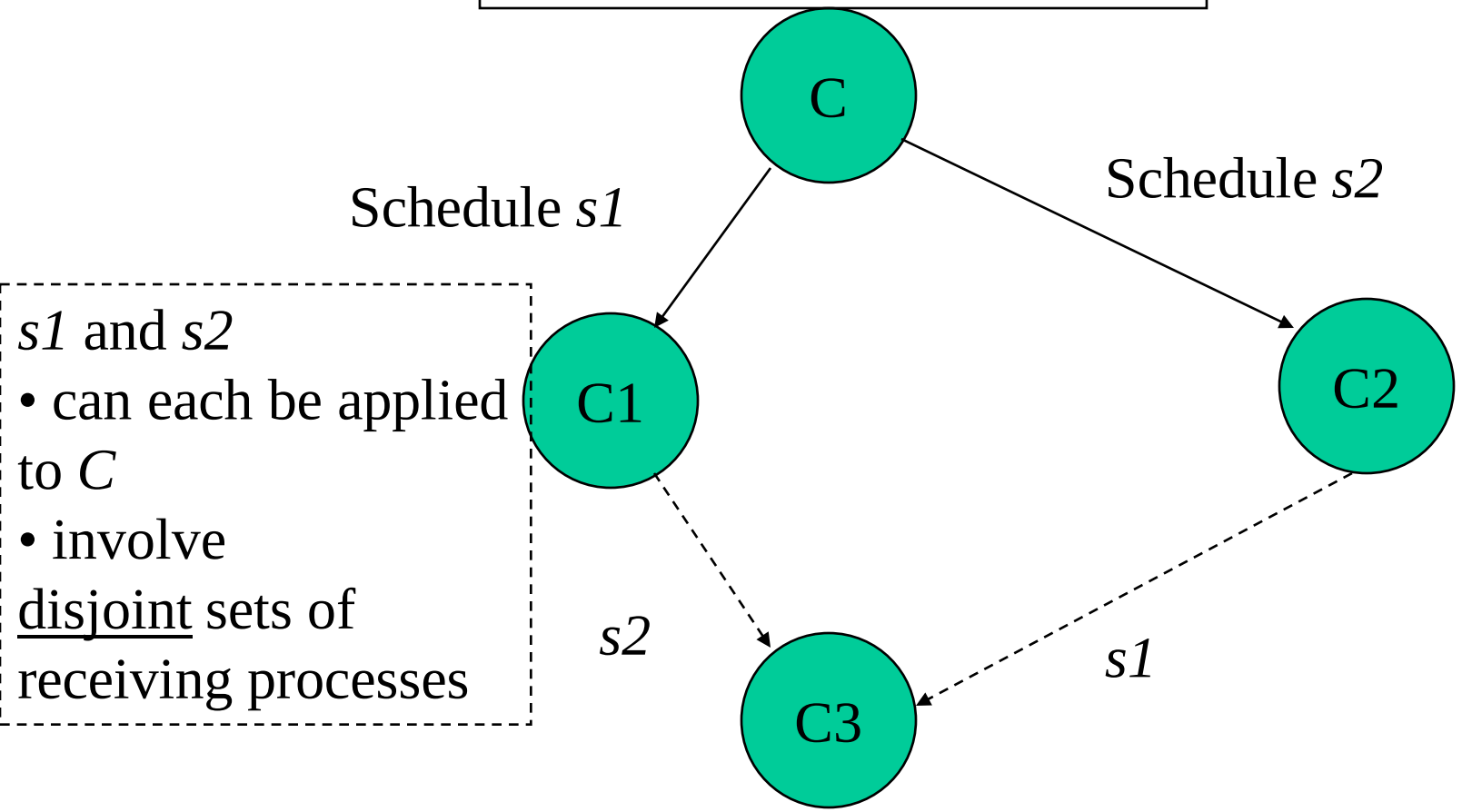
# Definitions

- **Internal State** of a process *p*
- **Configuration** *C*: = Collection of Internal States of each Process + content of global message buffer
  - Initial configuration:= configuration in which each process starts at an initial state and message buffer is empty
- Each **Event** *e=(p, m)* consists of
  - receipt of a message *m* by a process *p* and
  - processing of message *m,* and
  - sending out of all necessary messages by *p* (into the global message buffer)
    - *e(C )* = resulting configuration after event *e*, starting from configuration *C*;
    - Note: this event is different from the Lamport events
- **Schedule** *s*: sequence of events
  - e.g., *s=(e, e')* sequence of two events *e* and *e'*.
  - If *s* is finite, then *s(C)* , the resulting configuration, is said to be **reachable** from *C.*
  - A configuration reachable from some initial configuration is called **accessible.**
  - **Run:** schedule applied to a configuration


C
Configuration C
Event e'=(p',m')
C'
Event e''=(p'',m'')
C''
C
Schedule s=(e',e'')
C''
Equivalent

# **Lemma 1**(show properties about events, schedules, configurations)

**Schedules are commutative**




*s1* and *s2*
- can each be applied to *C*
- involve <u>disjoint</u> sets of receiving processes

# Easier Consensus Problem

Easier Consensus Problem: some process eventually sets $y_p$ to be **0** or **1**

Only one process crashes – we're free to choose which one

Consensus Protocol is partially correct if it satisfies two conditions

1. *No accessible configuration (config. reachable from an initial config.) has more than one decision value*
2. For each *v* in {0,1}, some accessible configuration (reachable from some initial state) has decision value *v*
   - avoids trivial solution to the consensus problem

Total correctness: partial correct with 1 failure + all admissible runs are deciding runs

# Main Goal: Show "No Consensus Protocol is totally correct in spite of one fault"

- Proof: By Contradiction (in two steps)
- Outline of the Proof:
  - Assume that **P** is a consensus protocol that is totally correct despite of one fault
  - Then show **circumstances** under which the protocol remains forever **indecisive** (i.e., has output value {b})
  - We will prove the impossibility result in **two steps:**
    - 1. Step: Argue that there **exists initial configuration** in which the decision is not already predetermined
    - 2. Step: Construct an **admissible run that avoids** ever taking a step that would commit the system to particular decision

# Valency Definition

- Let configuration *C* have a set of decision values *V* reachable from it
  - *C* is called bivalent if |V| = **2**
  - *C* is called univalent if |V| = **1**;
    - i.e., configuration C is said to be either 0-valent or 1-valent

- Bivalent means outcome is unpredictable

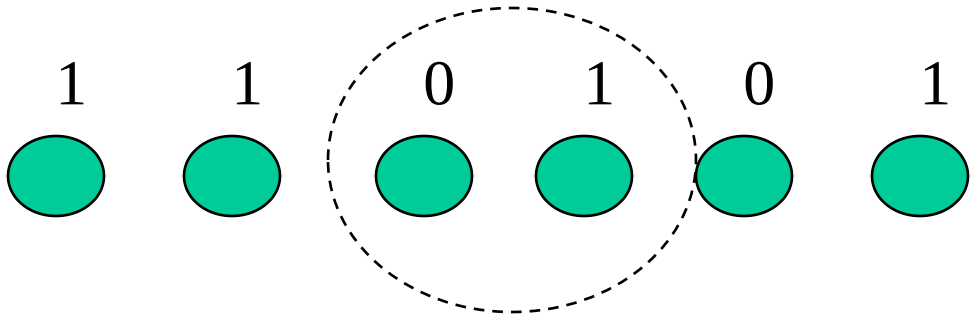
# What we will show

1. There exists an initial configuration that is bivalent (Lemma 2)
2. Starting from a bivalent configuration, there is always another bivalent configuration that is reachable (Lemma 3)

# Lemma 2

**Some initial configuration is bivalent**

- Proof: By Contradiction
- Suppose all initial configurations were predetermined either 0-valent or 1-valent.
- Place all initial configurations side-by-side, where **adjacent configurations** differ in initial $x_p$ value for *exactly one* process.



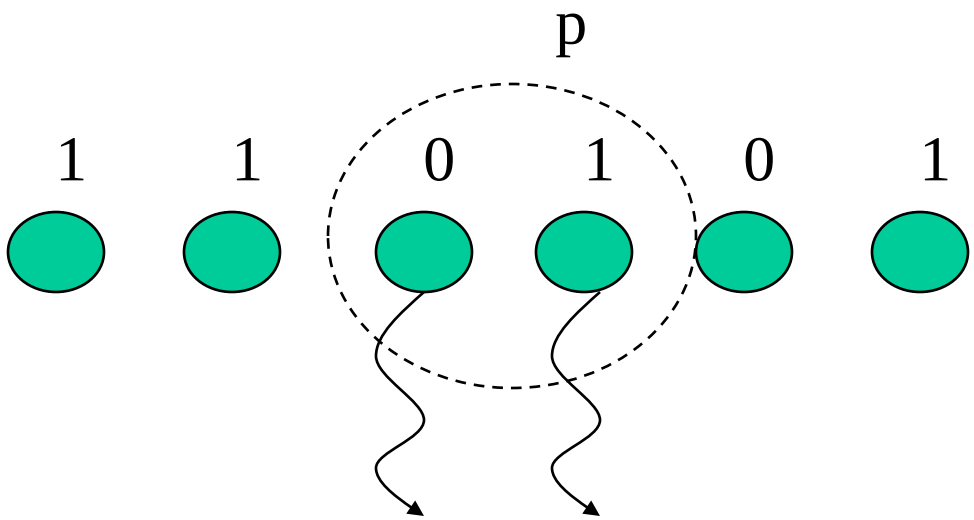

**Definition:** Two initial configurations are **adjacent** if they differ in the init value $x_p$ of a single process $p$.

- There ***has*** to be **some** adjacent pair of 1-valent and 0-valent configurations

# Lemma 2

**Some initial configuration is bivalent**

- There has to be **some** adjacent pair of 1-valent (*C1*) and 0-valent (*C0*) configurations
- Let the process *p* be the one with a different state across these two configurations *C0* and *C1*.
- Now consider the world where process *p* has crashed




Both these initial configurations are *indistinguishable*. But one gives a **0** decision value. The other gives a **1** decision value.

**So, both these initial configurations are bivalent when there is a failure**

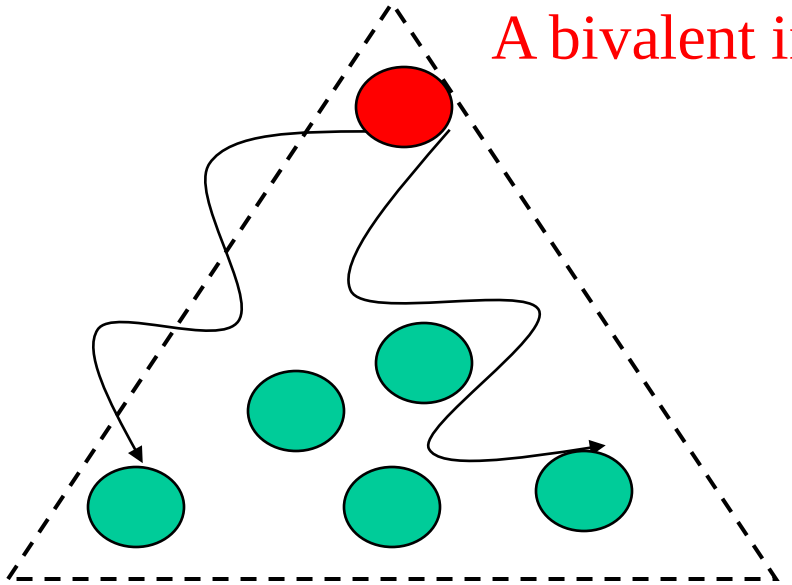
# What we will show

1. There exists an initial configuration that is bivalent (Lemma 2)
2. Starting from a bivalent configuration, there is always another bivalent configuration that is reachable

# Lemma 3

**Starting from a bivalent configuration, there is always another bivalent configuration that is reachable**

# Lemma 3



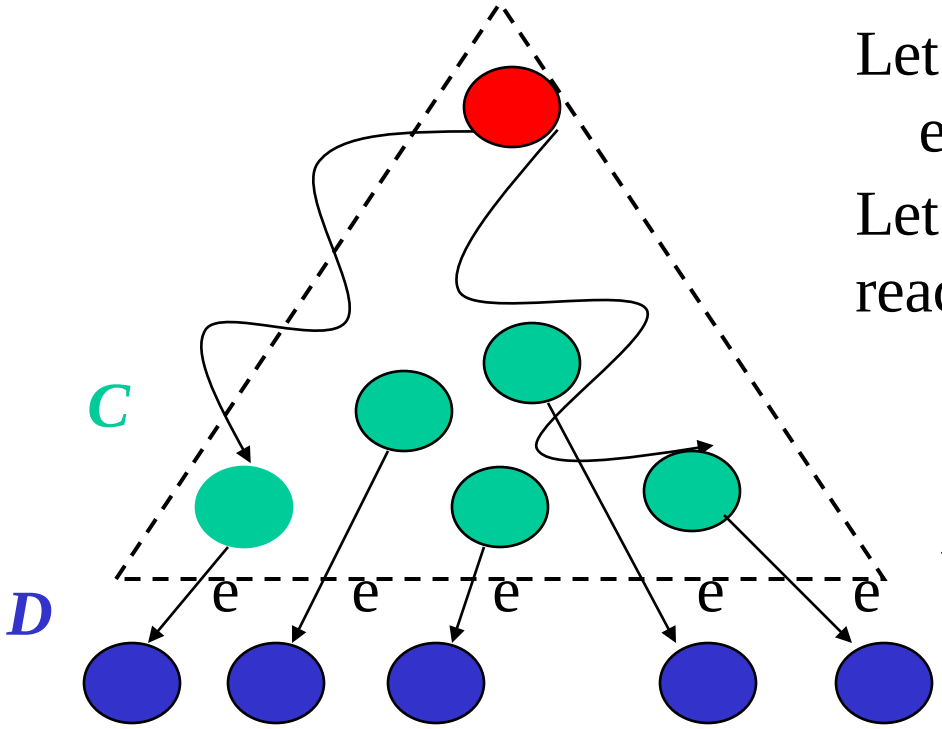

A bivalent initial configuration

Let $e=(p,m)$ be an applicable event to the initial configuration

Let ***C*** be the set of configurations reachable without applying e

# Lemma 3

A bivalent initial configuration

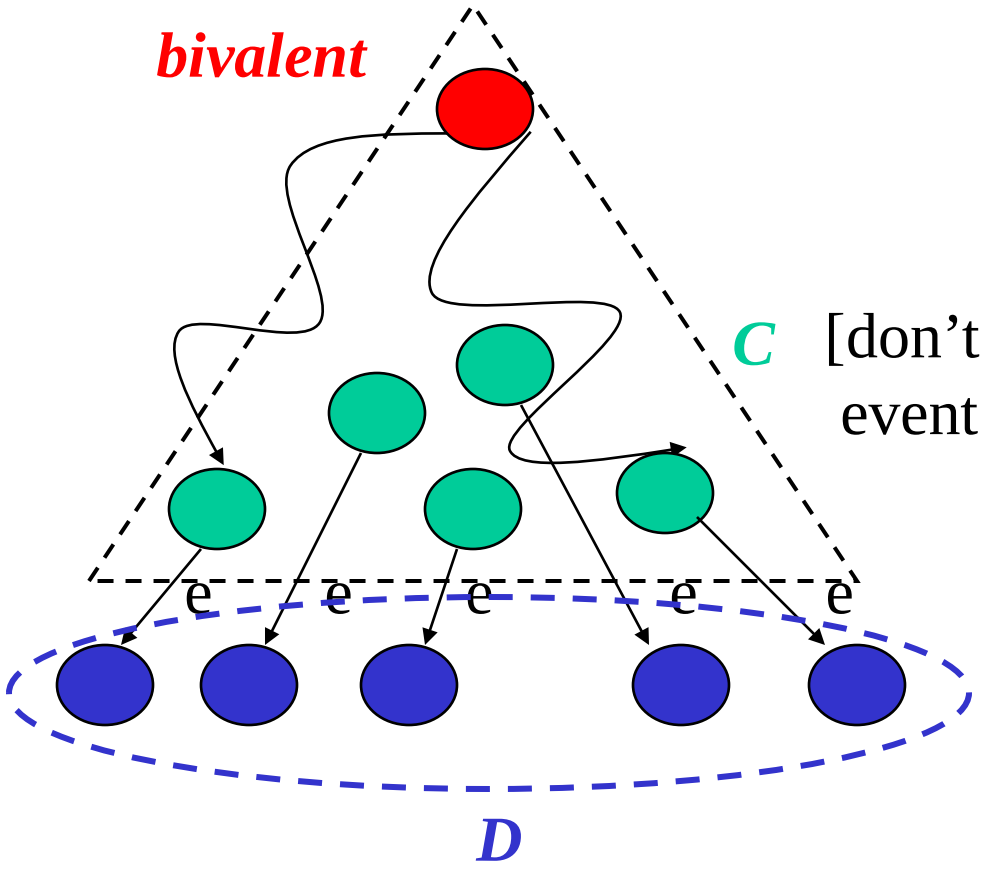



Let $e=(p,m)$ be an applicable event to the initial configuration

Let ***C*** be the set of configurations, reachable without applying $e$

Let ***D*** be the set of configurations obtained by applying single event $e$ to a configuration in ***C***

# Lemma 3



# Lemma 3

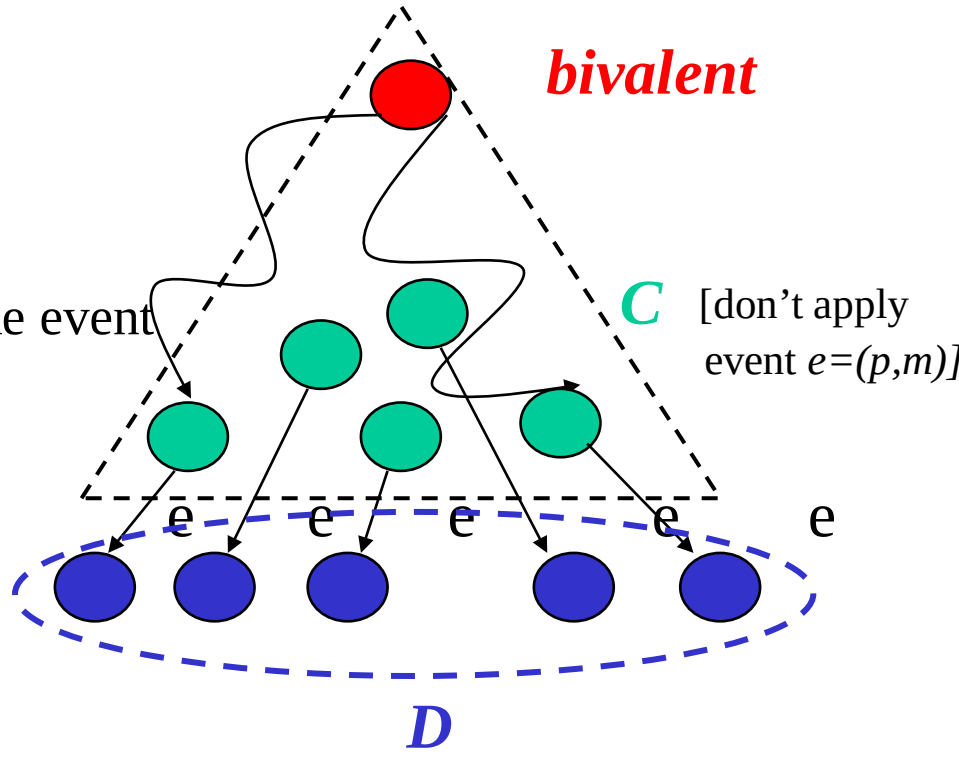
**Claim.** Set *D* contains a bivalent configuration

**Proof.** By contradiction.

- suppose ***D*** has only **0**- and **1**- valent states (and no bivalent ones)
- There are states *D0* and *D1* in ***D***, and *C0* and *C1* in ***C*** such that
  - *D0* is **0**-valent, *D1* is **1**-valent
  - *D0=e(C0 )* followed by *e=(p,m)*
  - *D1=e(C1)* followed by *e=(p,m)*
  - And *C1 = e'(C0)* followed by some event *e'=(p',m')*

***THEN By Lemma 1 follows:***

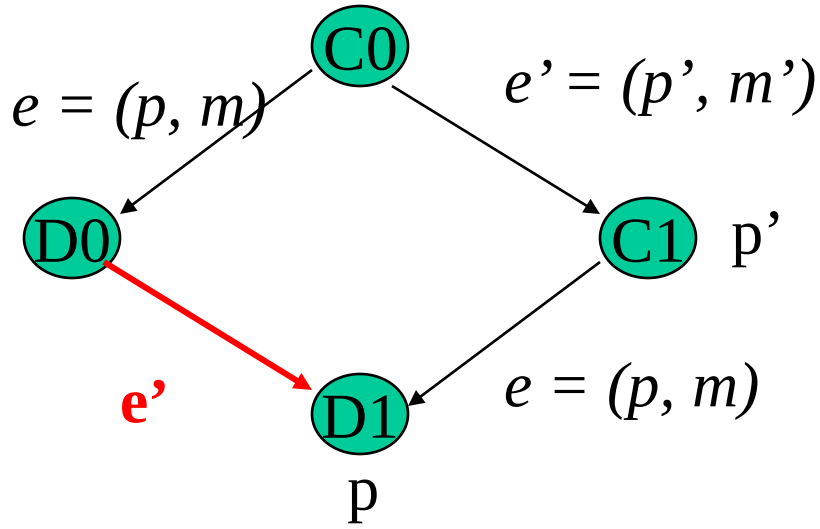
***D1 = e'(D0)***

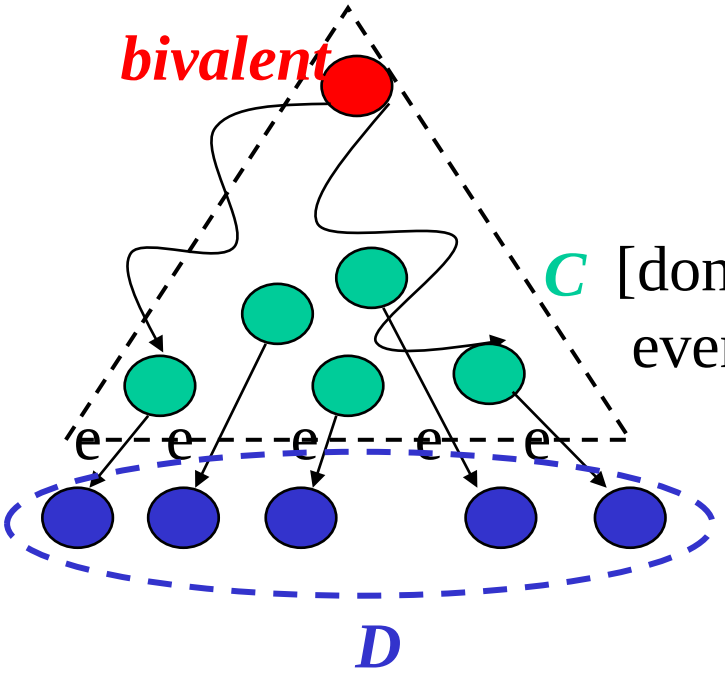


# Lemma 3

**Proof.** (contd.)

- Case I: p' is not p
- Case II: p' same as p



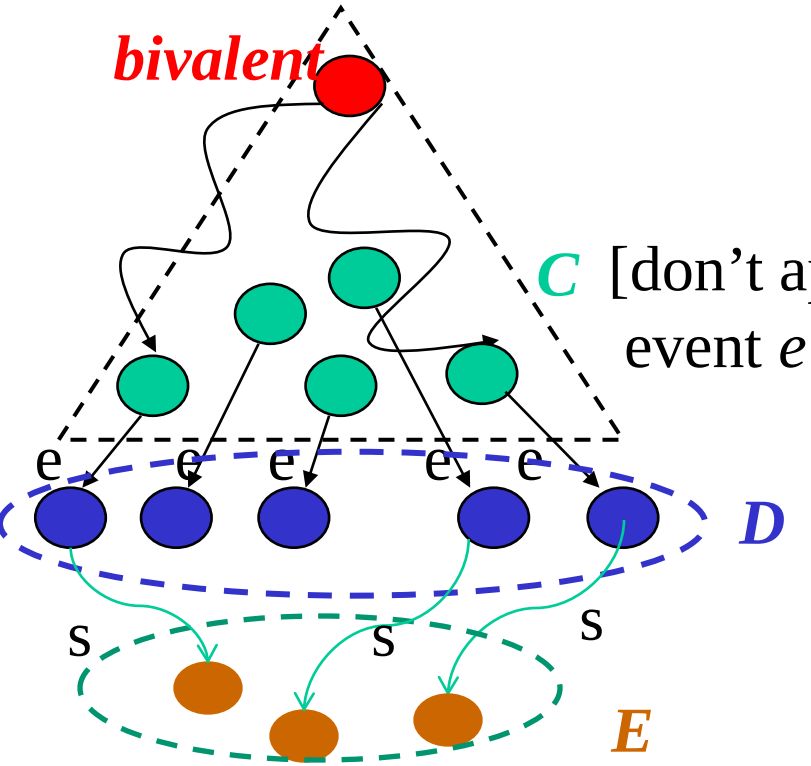

*C* [don't apply event *e=(p,m)*]

- From C1 follows D1 is 1-valent
- From Lemma 1 follows D1 = e'(D0), successor of 0-valent configuration is 0-valent hence contradiction
- D contains a bivalent configuration

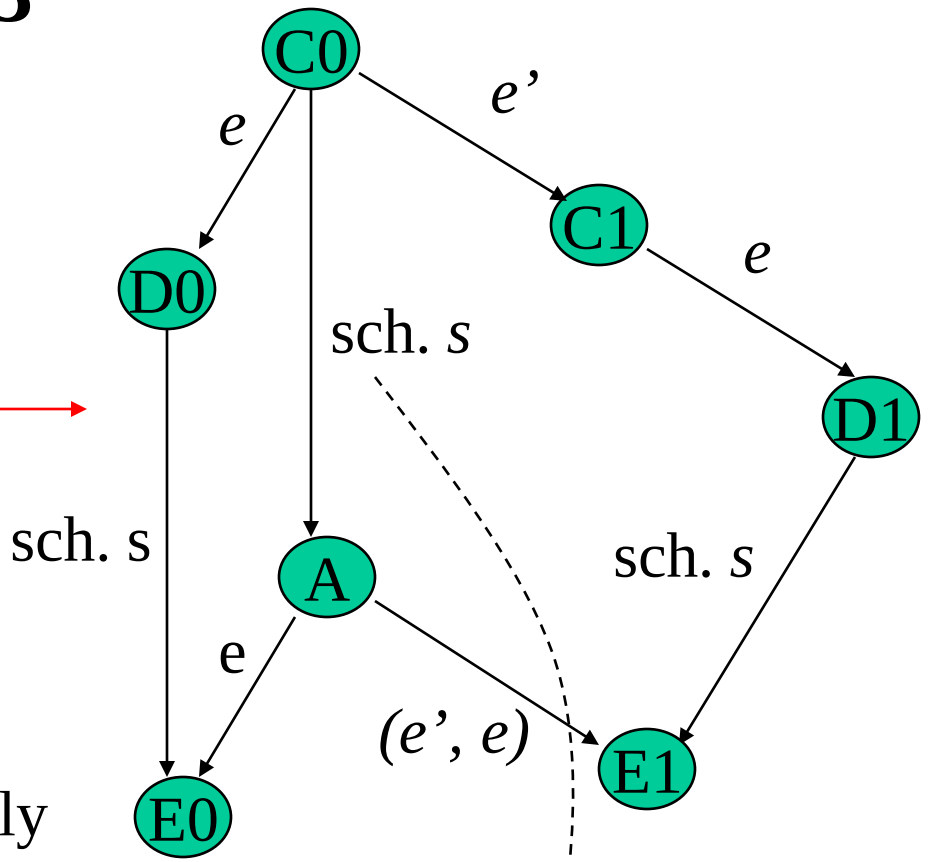
# Lemma 3

**Proof.** (contd.)

- Case I: p' is not p
- Case II: p' same as p




*C* [don't apply event *e=(p,m)*]




sch. *s*
- finite
- deciding run from *C0*
- *p* takes no steps

But A is then bivalent!

# Putting it all Together

- Lemma 2: There exists an initial configuration that is bivalent
- Lemma 3: Starting from a bivalent configuration, there is always another bivalent configuration that is reachable

- Theorem (Impossibility of Consensus): There is always a run of events in an asynchronous distributed system (given any algorithm) such that the group of processes never reaches consensus (i.e., always stays bivalent)
  - "The devil's advocate always has a way out"

# Why is Consensus Important? –

Many problems in distributed systems are equivalent to *(or harder than)* consensus!

- **Agreement**, e.g., on an integer (harder than consensus, since it can be used to solve consensus) **is impossible!**
- **Leader election is impossible!**
  - A leader election algorithm can be designed using a given consensus algorithm as a black box
  - A consensus protocol can be designed using a given leader election algorithm as a black box
- **Accurate Failure Detection is impossible!**
  - Should I mark a process that has not responded for the last 60 seconds as failed? (It might just be very, very, slow)

# Why is Consensus Important?

- The impossibility of consensus means there exists ***no*** perfect solutions to ***any*** of the above problems in **asynchronous system models**
  - In an asynchronous system, there is no perfect algorithm for either failure detection, or leader election, or agreement
- How do we get around this? One way is to design ***Probabilistic Algorithms***

# Summary

- Consensus Problem
  - Agreement in distributed systems
  - Solution exists in synchronous system model (e.g., supercomputer)
  - **Impossible to solve in an asynchronous system** (e.g., Internet, Web)
    - Key idea: with one process failure, there are circumstances under which the protocol remains forever indecisive .
  - FLP impossibility proof

# Before you go…

- Next lecture - Failure detectors: Read Sections 12.1 and 2.3.2
- H2 is out