

CS 425 / ECE 428  
Distributed Systems  
Fall 2015

Indranil Gupta (Indy)

Oct 13, 2015

*Lecture 13: Impossibility of Consensus*

All slides © IG

# GIVE IT A THOUGHT

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

The fault does not lie with the companies themselves, or the worthlessness of humanity.

The fault lies in the impossibility of consensus



# WHAT IS COMMON TO ALL OF THESE?

A group of servers attempting:

- Make sure that all of them receive the same updates in the same order as each other
- To keep their own local lists where they know about each other, and when anyone leaves or fails, everyone is updated simultaneously
- Elect a leader among them, and let everyone in the group know about it
- To ensure mutually exclusive (one process at a time only) access to a critical resource like a file



# WHAT IS COMMON TO ALL OF THESE?

A group of servers attempting:

- Make sure that all of them receive the same updates in the same order as each other [Reliable Multicast]
- To keep their own local lists where they know about each other, and when anyone leaves or fails, everyone is updated simultaneously [Membership/Failure Detection]
- Elect a leader among them, and let everyone in the group know about it [Leader Election]
- To ensure mutually exclusive (one process at a time only) access to a critical resource like a file [Mutual Exclusion]



# So WHAT IS COMMON?

- Let's call each server a “process” (think of the daemon at each server)
- All of these were groups of processes attempting to *coordinate* with each other and reach *agreement* on the value of something
  - The ordering of messages
  - The up/down status of a suspected failed process
  - Who the leader is
  - Who has access to the critical resource
- All of these are related to the *Consensus* problem



# WHAT IS CONSENSUS?

Formal problem statement

- $N$  processes
- Each process  $p$  has
  - input variable  $x_p$  : initially either 0 or 1
  - output variable  $y_p$  : initially  $b$  (can be changed only once)
- **Consensus problem**: design a protocol so that at the end, either:
  1. All processes set their output variables to 0 (all-0's)
  2. Or All processes set their output variables to 1 (all-1's)



# WHAT IS CONSENSUS? (2)

- Every process contributes a value
- *Goal is to have all processes decide same (some) value*
  - Decision once made can't be changed
- There might be other constraints
  - Validity = if everyone proposes same value, then that's what's decided
  - Integrity = decided value must have been proposed by some process
  - Non-triviality = there is at least one initial system state that leads to each of the all-0's or all-1's outcomes



# WHY IS IT IMPORTANT?

- Many problems in distributed systems are *equivalent to (or harder than)* consensus!
  - Perfect Failure Detection
  - Leader election (select exactly one leader, and every alive process knows about it)
  - Agreement (harder than consensus)
- So consensus is a very important problem, and solving it would be really useful!
- So, is there a solution to Consensus?



# TWO DIFFERENT MODELS OF DISTRIBUTED SYSTEMS

- Synchronous System Model and Asynchronous System Model
- Synchronous Distributed System
  - Each message is received within bounded time
  - Drift of each process' local clock has a known bound
  - Each step in a process takes  $lb < \text{time} < ub$

*E.g., A collection of processors connected by a communication bus, e.g., a Cray supercomputer or a multicore machine*



# ASYNCHRONOUS SYSTEM MODEL

- **Asynchronous** Distributed System
  - No bounds on process execution
  - The drift rate of a clock is arbitrary
  - No bounds on message transmission delays

*E.g., The Internet is an asynchronous distributed system, so are ad-hoc and sensor networks*
- *This is a more general (and thus challenging) model than the synchronous system model. A protocol for an asynchronous system will also work for a synchronous system (but not vice-versa)*

# POSSIBLE OR NOT

- In the synchronous system model
  - Consensus is solvable
- In the asynchronous system model
  - Consensus is impossible to solve
  - Whatever protocol/algorithm you suggest, there is always a worst-case possible execution (with failures and message delays) that prevents the system from reaching consensus
  - Powerful result (*see the FLP proof elsewhere in this course*)
  - Subsequently, safe or probabilistic solutions have become quite popular to consensus or related problems.



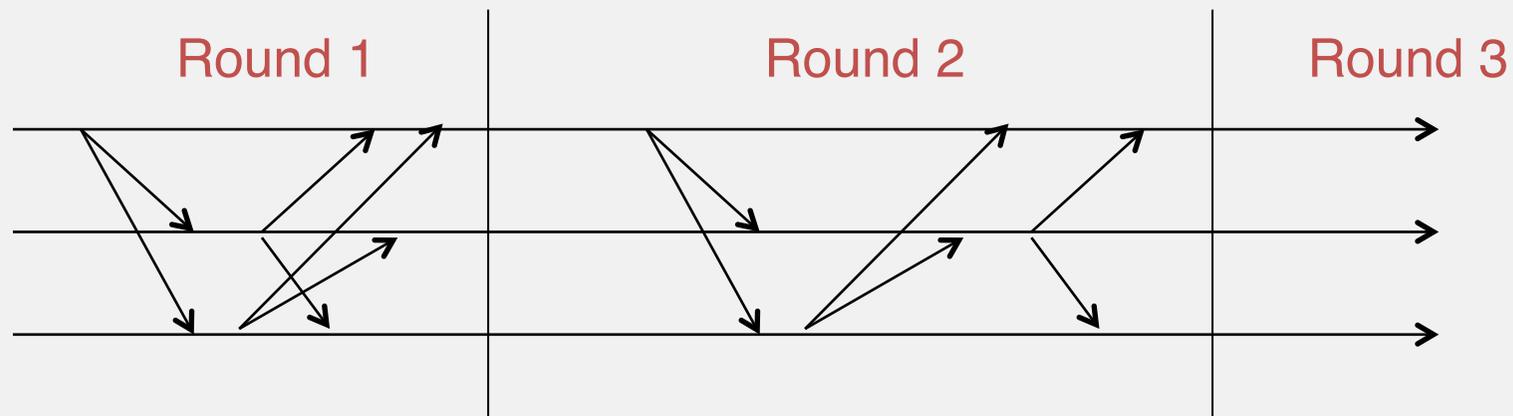
# LET'S TRY TO SOLVE CONSENSUS!

- Uh, what's the **system model?**  
(assumptions!)
- **Synchronous system:** bounds on
  - Message delays
  - Upper bound on clock drift rates
  - Max time for each process stepe.g., multiprocessor (common clock across processors)
- **Processes can fail by stopping (crash-stop or crash failures)**



# CONSENSUS IN SYNCHRONOUS SYSTEMS

- For a system with at most  $f$  processes crashing
  - All processes are synchronized and operate in “rounds” of time. Round length  $\gg$  max transmission delay.
  - the algorithm proceeds in  $f+1$  rounds (with timeout), using reliable communication to all members
  - $Values^r_i$ : the set of proposed values known to  $p_i$  at the beginning of round  $r$ .



# CONSENSUS IN SYNCHRONOUS SYSTEM

Possible to achieve!

- For a system with at most  $f$  processes crashing
  - All processes are synchronized and operate in “rounds” of time
  - the algorithm proceeds in  $f+1$  rounds (with timeout), using reliable communication to all members. Round length  $\gg$  max transmission delay.
  - $Values^r_i$ : the set of proposed values known to  $p_i$  at the beginning of round  $r$ .
- Initially  $Values^0_i = \{\}$  ;  $Values^1_i = \{v_i\}$ 
  - for round = 1 to  $f+1$  do
    - multicast** ( $Values^r_i - Values^{r-1}_i$ ) // iterate through processes, send each a message
    - $Values^{r+1}_i \leftarrow Values^r_i$
    - for each  $V_j$  received
      - $Values^{r+1}_i = Values^{r+1}_i \cup V_j$
    - end
  - end
  - $d_i = \mathbf{minimum}(Values^{f+1}_i)$



# WHY DOES THE ALGORITHM WORK?

- After  $f+1$  rounds, all non-faulty processes would have received the same set of Values. Proof by contradiction.
- Assume that two non-faulty processes, say  $p_i$  and  $p_j$ , differ in their final set of values (i.e., after  $f+1$  rounds)
- Assume that  $p_i$  possesses a value  $v$  that  $p_j$  does not possess.
  - $p_i$  must have received  $v$  in the **very last** round
    - Else,  $p_i$  would have sent  $v$  to  $p_j$  in that last round
  - So, in the last round: a third process,  $p_k$ , must have sent  $v$  to  $p_i$ , but then crashed before sending  $v$  to  $p_j$ .
  - Similarly, a fourth process sending  $v$  in the **last-but-one round** must have crashed; otherwise, both  $p_k$  and  $p_j$  should have received  $v$ .
  - Proceeding in this way, we infer at least one (unique) crash in each of the preceding rounds.
  - This means a total of  $f+1$  crashes, while we have assumed at most  $f$  crashes can occur => contradiction.



# CONSENSUS IN AN ASYNCHRONOUS SYSTEM

- Impossible to achieve!
- 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

**Asynchronous system:** All message delays and processing delays can be arbitrarily long or short.

**Consensus:**

- Each process  $p$  has a **state**
  - program counter, registers, stack, local variables
  - input register  $x_p$  : initially either 0 or 1
  - output register  $y_p$  : initially  $b$  (undecided)
- Consensus Problem: design a protocol so that either
  - all processes set their output variables to 0 (all-0's)
  - Or all processes set their output variables to 1 (all-1's)
  - Non-triviality: at least one initial system state leads to each of the above two outcomes

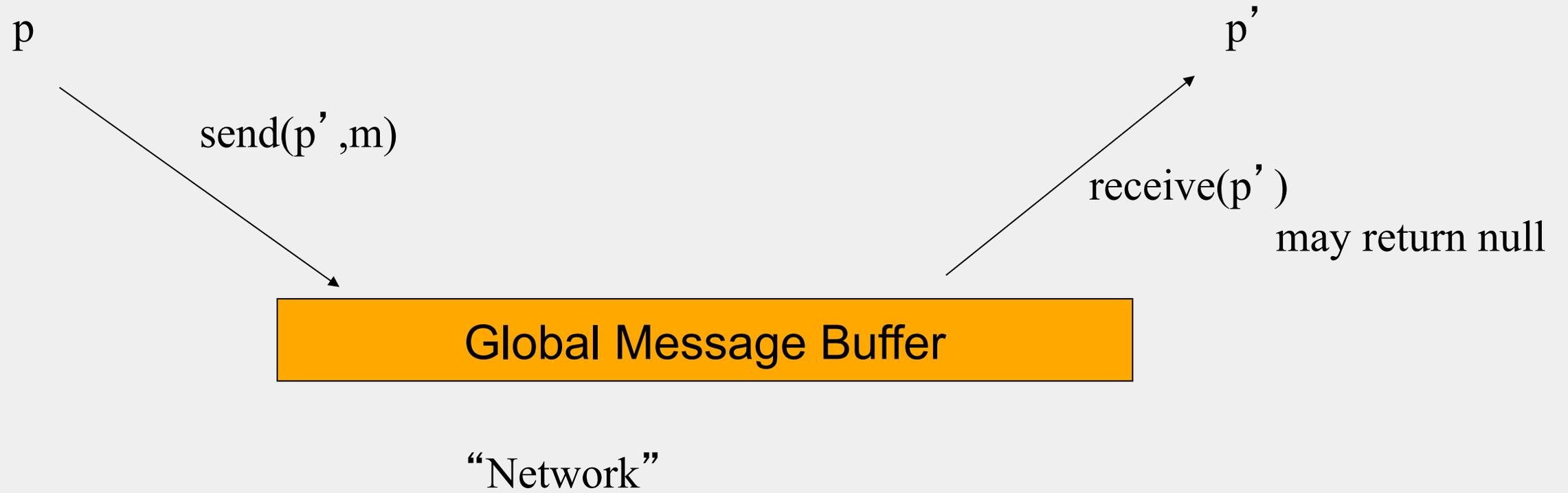


# PROOF SETUP

- For impossibility proof, OK to consider
  1. more restrictive system model, and
  2. easier problem
    - Why is this is ok?



# NETWORK

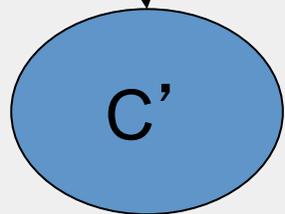


# STATES

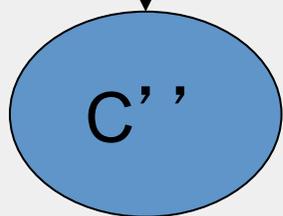
- State of a process
- **Configuration=global state.** Collection of states, one for each process; alongside state of the global buffer.
- Each **Event** (different from Lamport events) is atomic and consists of three steps
  - receipt of a message by a process (say p)
  - processing of message (may change recipient's state)
  - sending out of all necessary messages by p
- **Schedule:** sequence of events



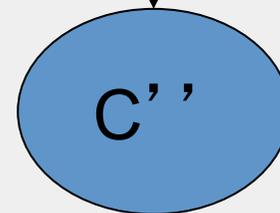
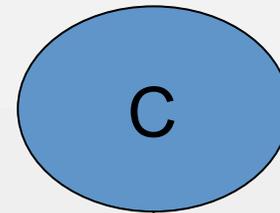
Event  $e' = (p', m')$



Event  $e'' = (p'', m'')$



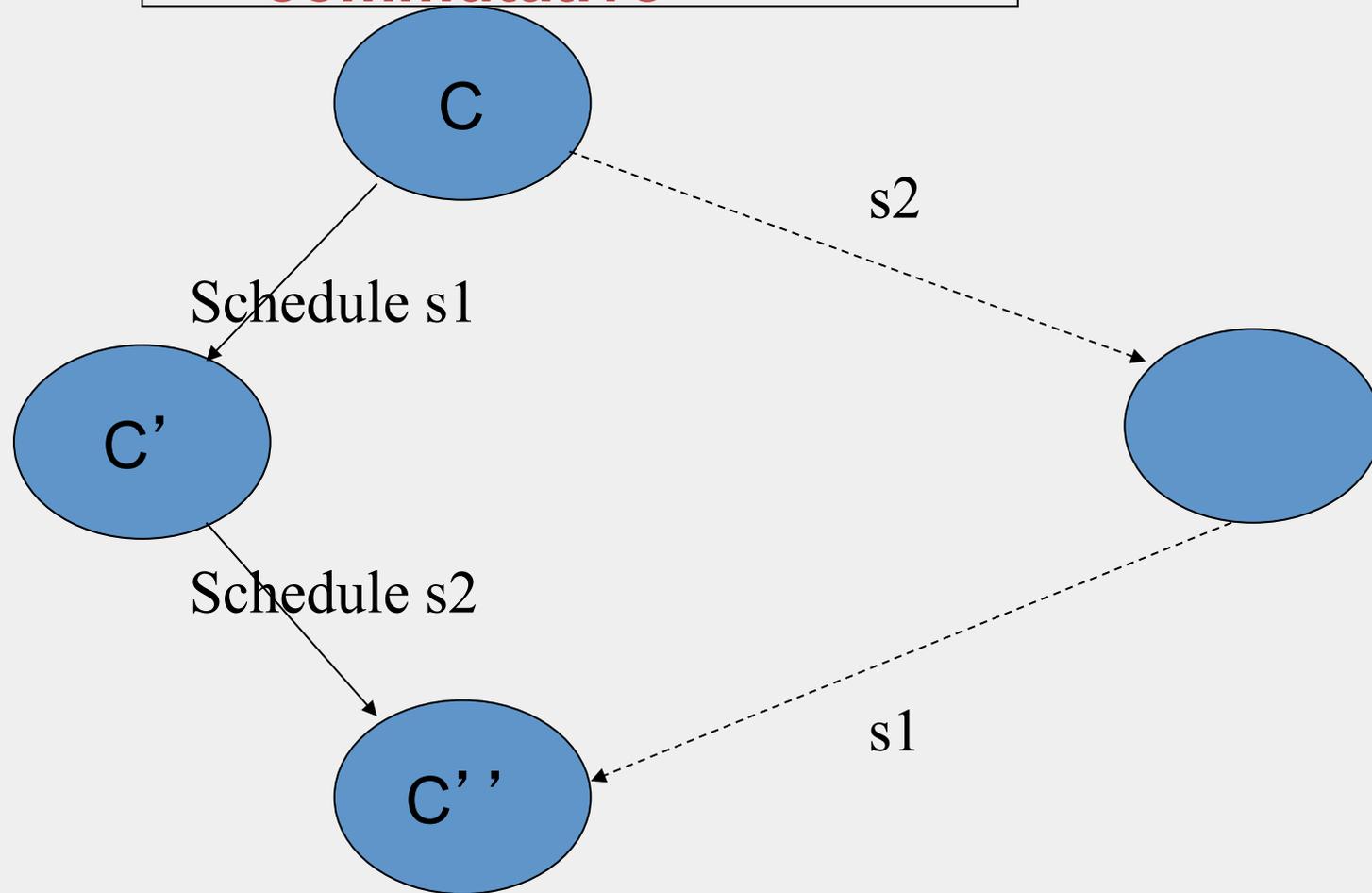
Schedule  $s = (e', e'')$



Equivalent

# LEMMA 1

Disjoint schedules are commutative



s1 and s2 involve disjoint sets of receiving processes, and are each applicable on C

# 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



# EASIER CONSENSUS PROBLEM

- Let config.  $C$  have a set of decision values  $V$  reachable from it
  - If  $|V| = 2$ , config.  $C$  is bivalent
  - If  $|V| = 1$ , config.  $C$  is 0-valent or 1-valent, as is the case
- **Bivalent** means **outcome is unpredictable**



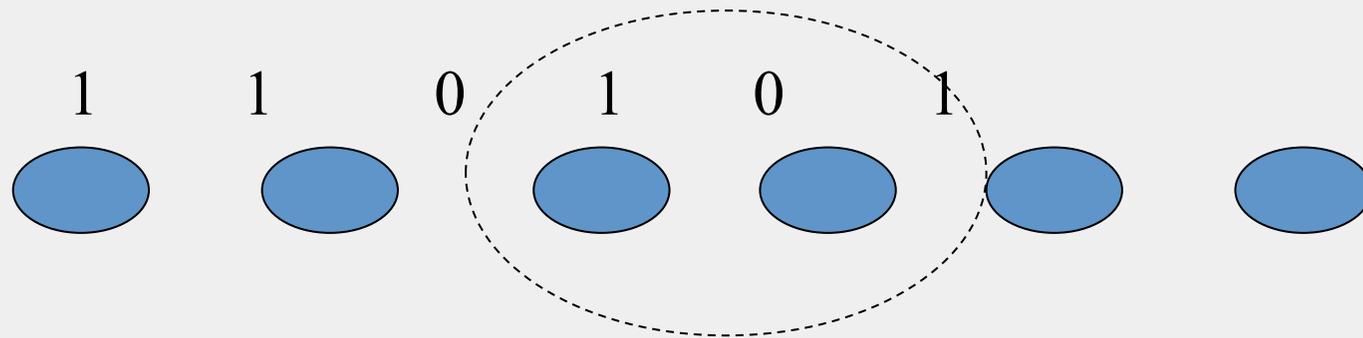
# WHAT THE FLP PROOF SHOWS

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

# LEMMA 2

Some initial configuration is bivalent

- Suppose all initial configurations were either 0-valent or 1-valent.
- If there are  $N$  processes, there are  $2^N$  possible initial configurations
- Place all configurations side-by-side (in a lattice), where adjacent configurations differ in initial xp value for exactly one process.

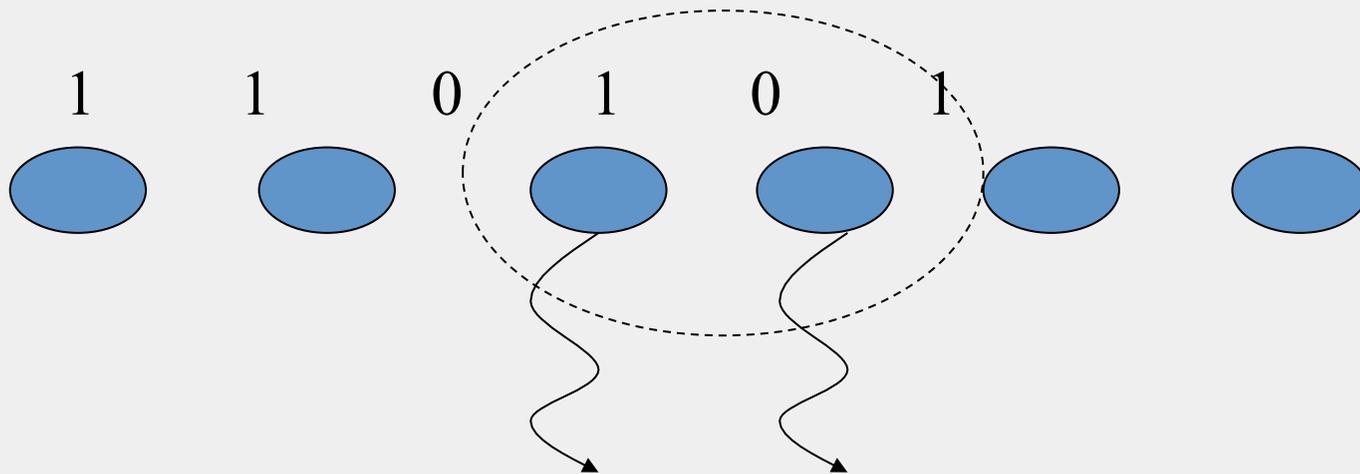


- There has to be **some** adjacent pair of 1-valent and 0-valent configs.

# LEMMA 2

Some initial configuration is bivalent

- There has to be **some** adjacent pair of 1-valent and 0-valent configs.
- Let the process  $p$ , that has a different state across these two configs., be the process that has crashed (i.e., is silent throughout)



Both initial configs. will lead to the same config. for the same sequence of events

Therefore, both these initial configs. are bivalent when there is such a failure

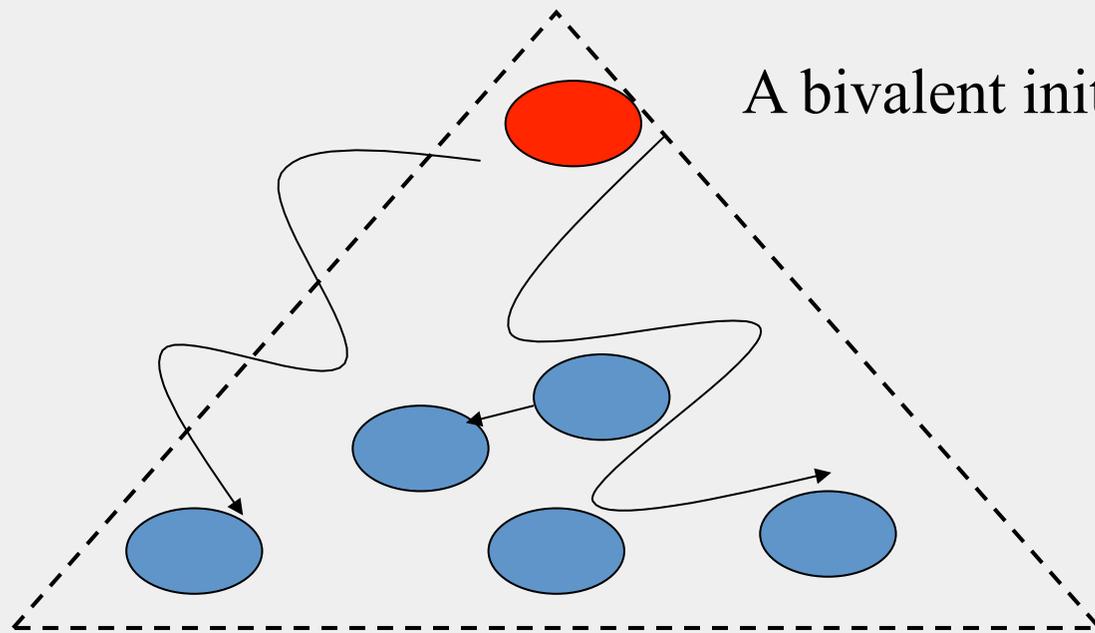
# WHAT WE'LL SHOW

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

# LEMMA 3

Starting from a bivalent config., there is always another bivalent config. that is reachable

# LEMMA 3

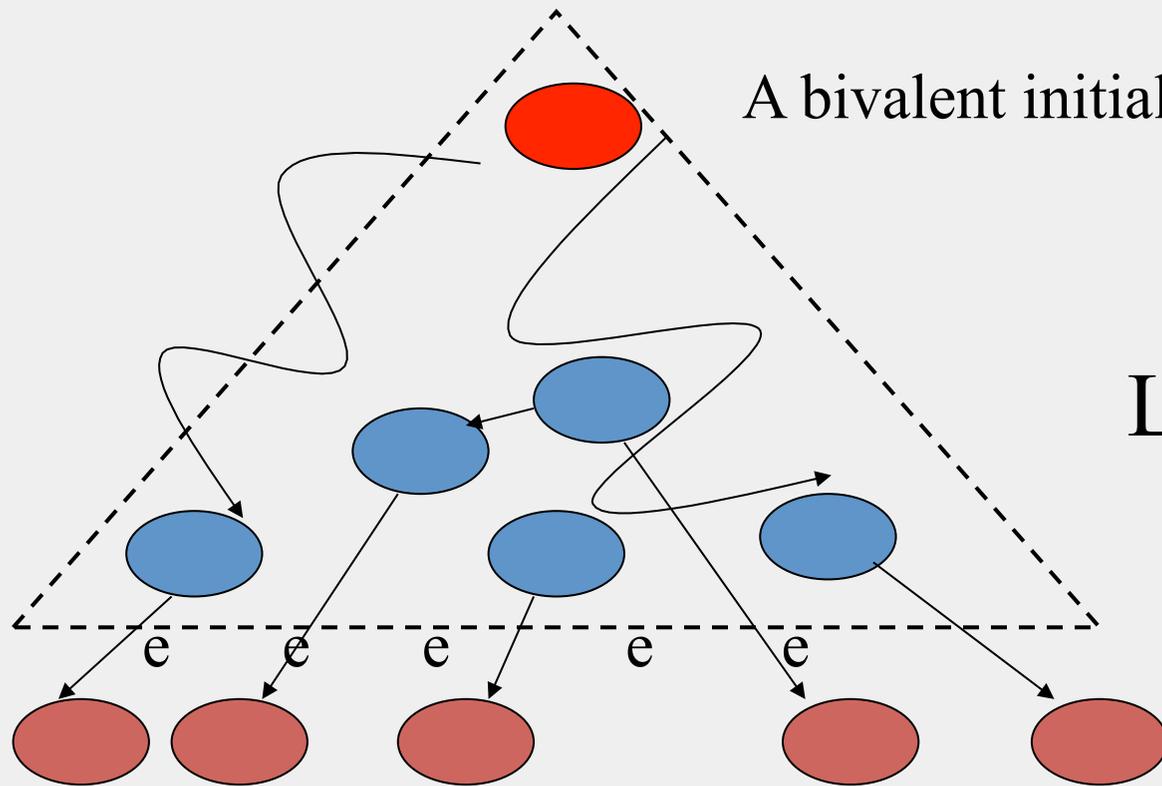


A bivalent initial config.

let  $e=(p,m)$  be some event  
applicable to the initial config.

Let  $C$  be the set of configs. reachable  
**without** applying  $e$

# LEMMA 3



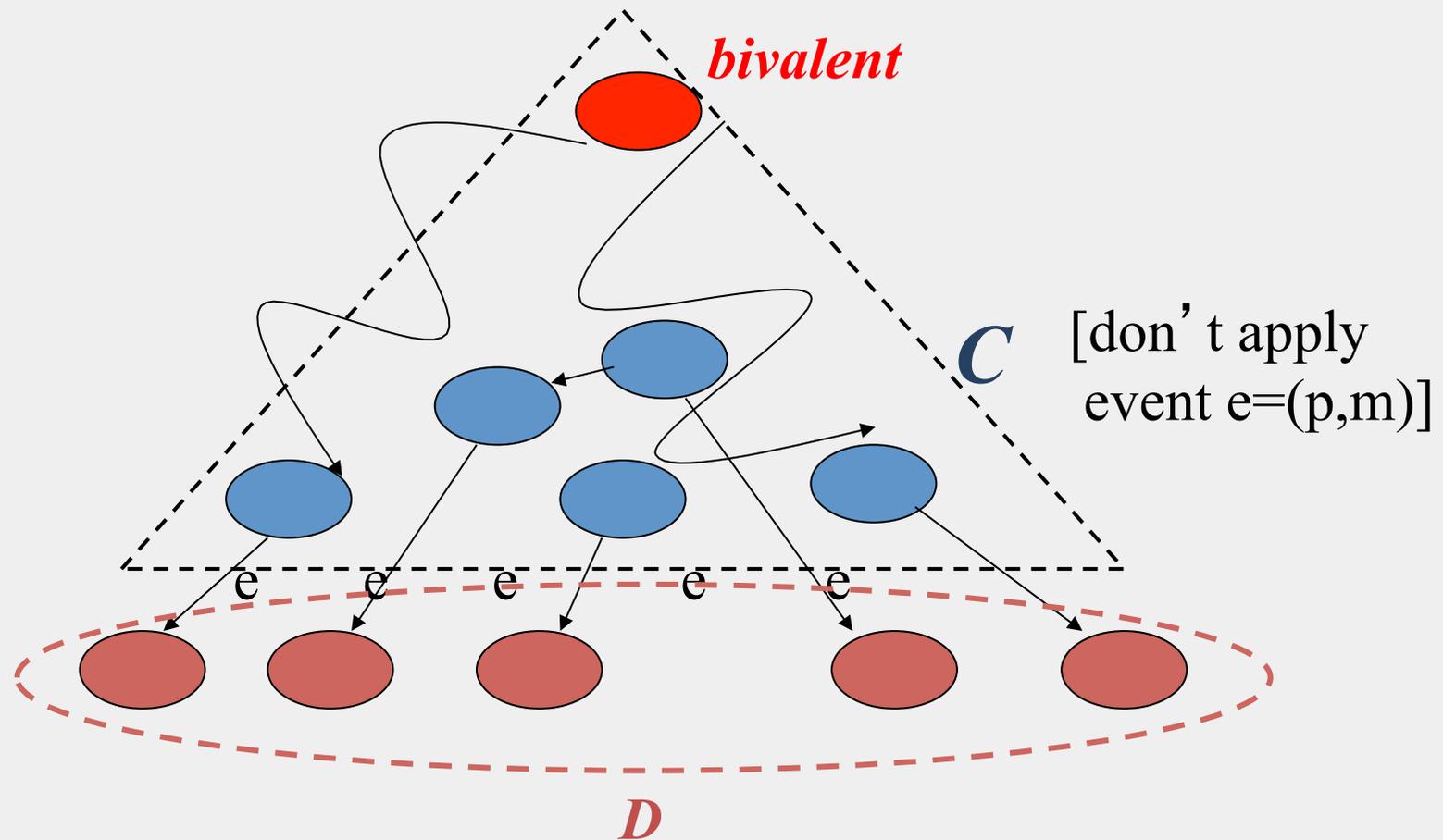
A bivalent initial config.

let  $e=(p,m)$  be some event  
applicable to the initial config.

Let  $C$  be the set of configs. reachable  
**without** applying  $e$

Let  $D$  be the set of configs.  
obtained by **applying**  $e$  to some  
config. in  $C$

# LEMMA 3



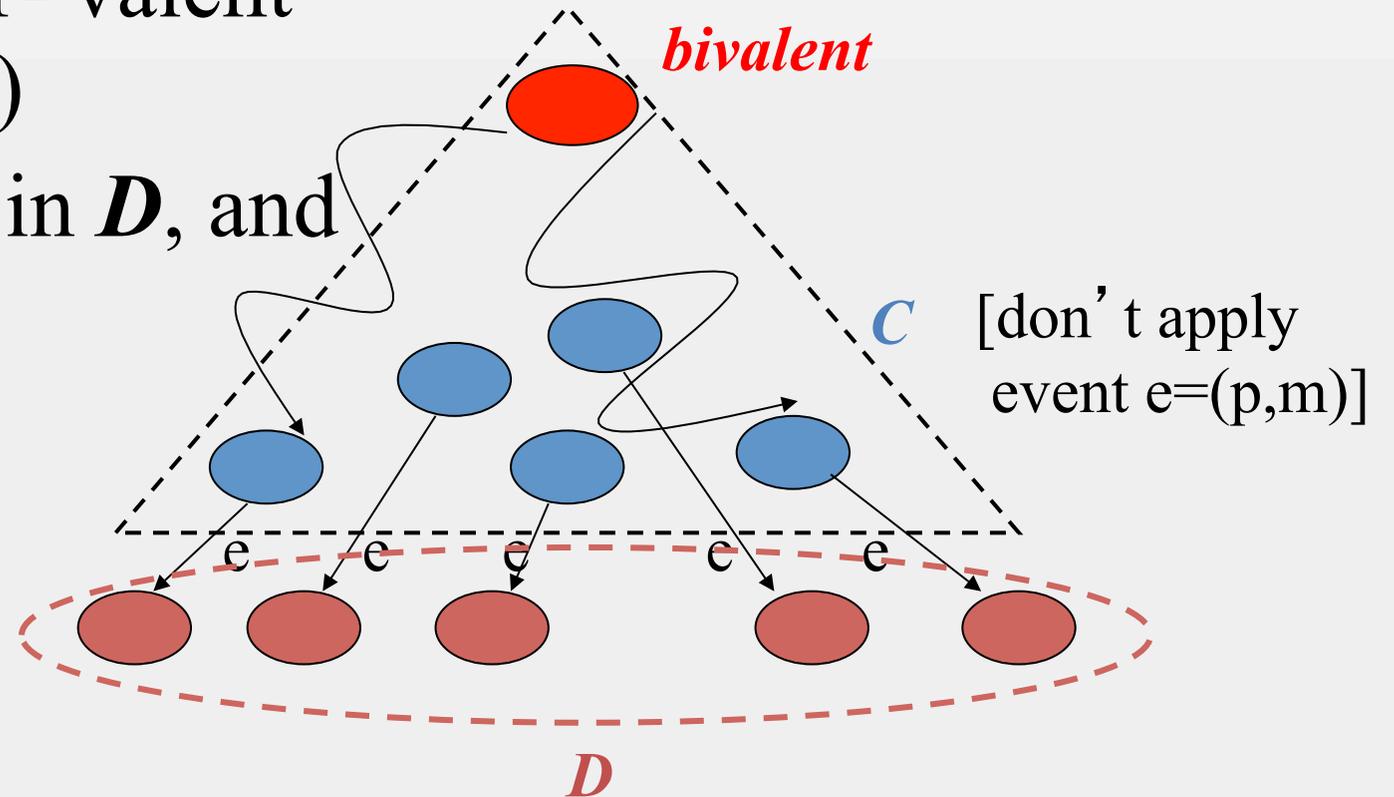
**Claim.** Set  $D$  contains a bivalent config.

**Proof.** By contradiction. That is,  
suppose  $D$  has only 0- and 1- valent states (and no bivalent ones)

- There are states  $D_0$  and  $D_1$  in  $D$ , and  $C_0$  and  $C_1$  in  $C$  such that

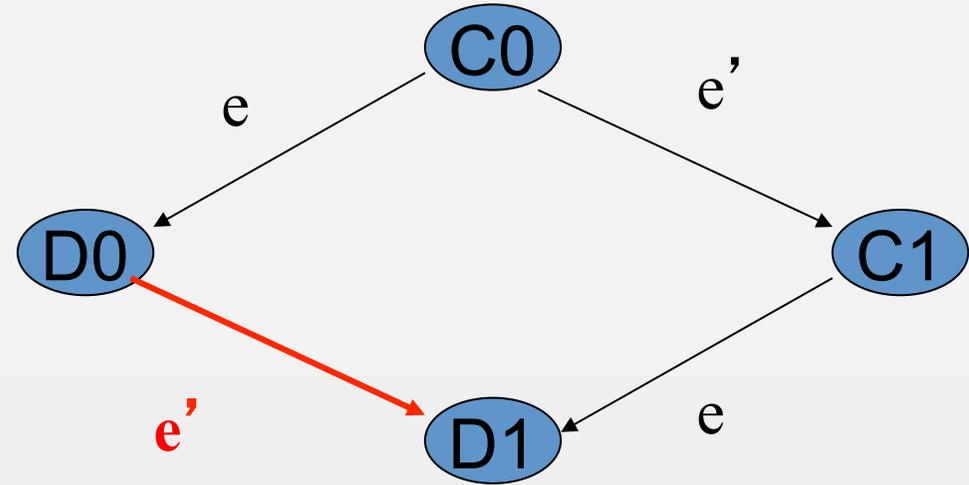
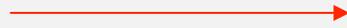
- $D_0$  is 0-valent,  $D_1$  is 1-valent
- $D_0 = C_0$  foll. by  $e = (p, m)$
- $D_1 = C_1$  foll. by  $e = (p, m)$
- And  $C_1 = C_0$  followed by some event  $e' = (p', m')$

(why?)



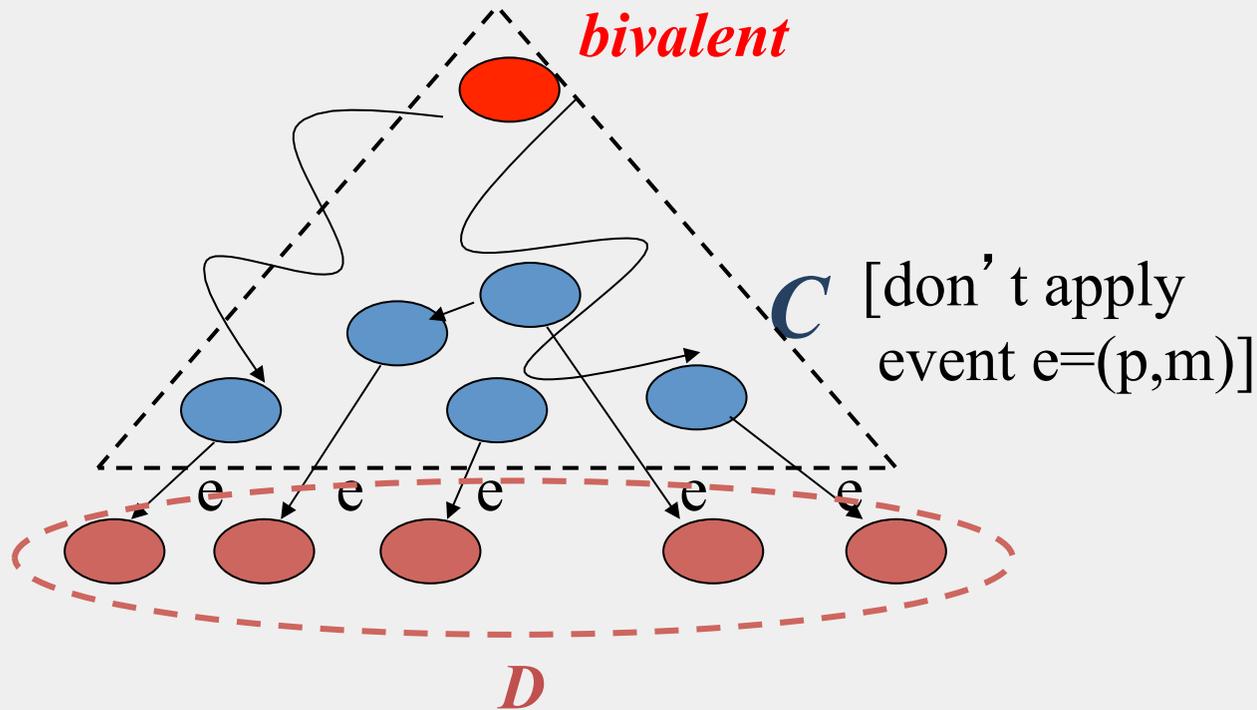
**Proof.** (contd.)

- Case I:  $p'$  is not  $p$



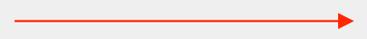
- Case II:  $p'$  same as  $p$

Why? (Lemma 1)  
But D0 is then bivalent!

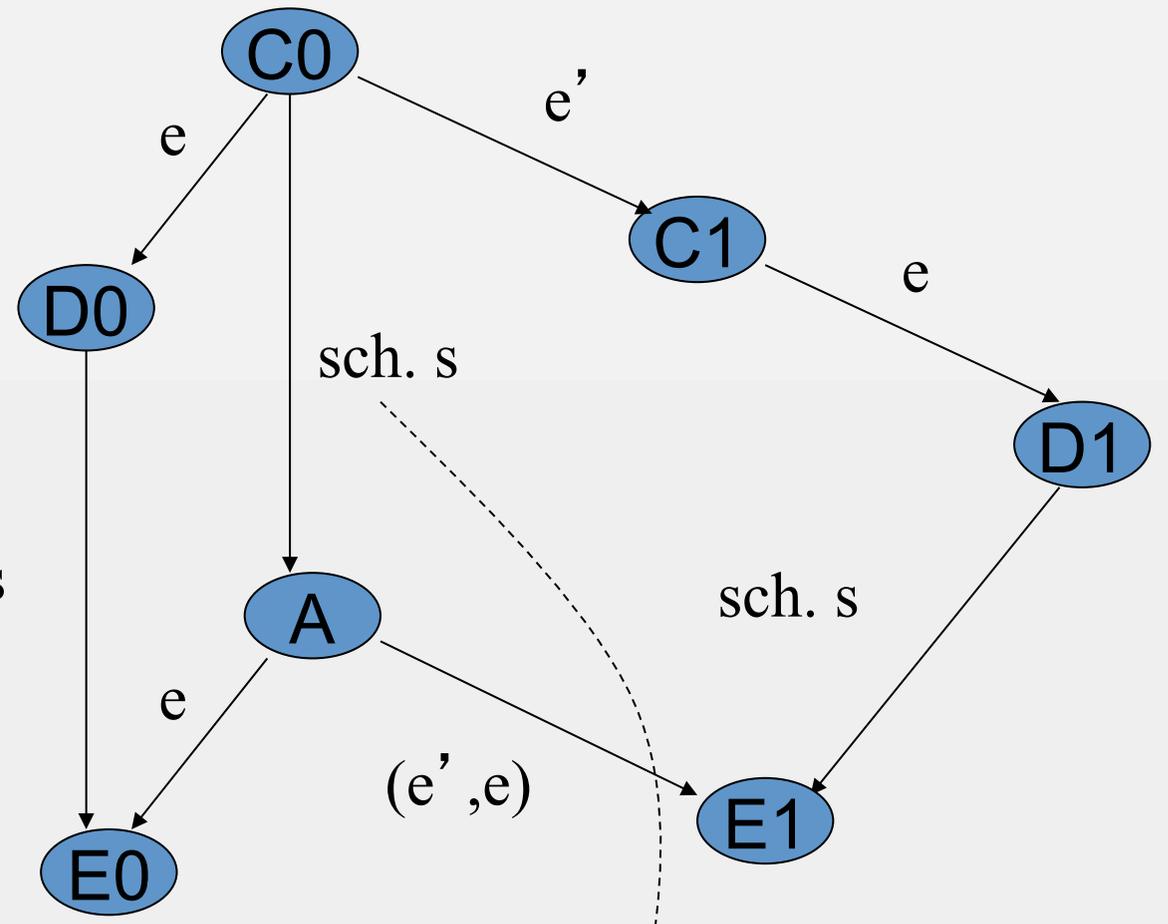


**Proof.** (contd.)

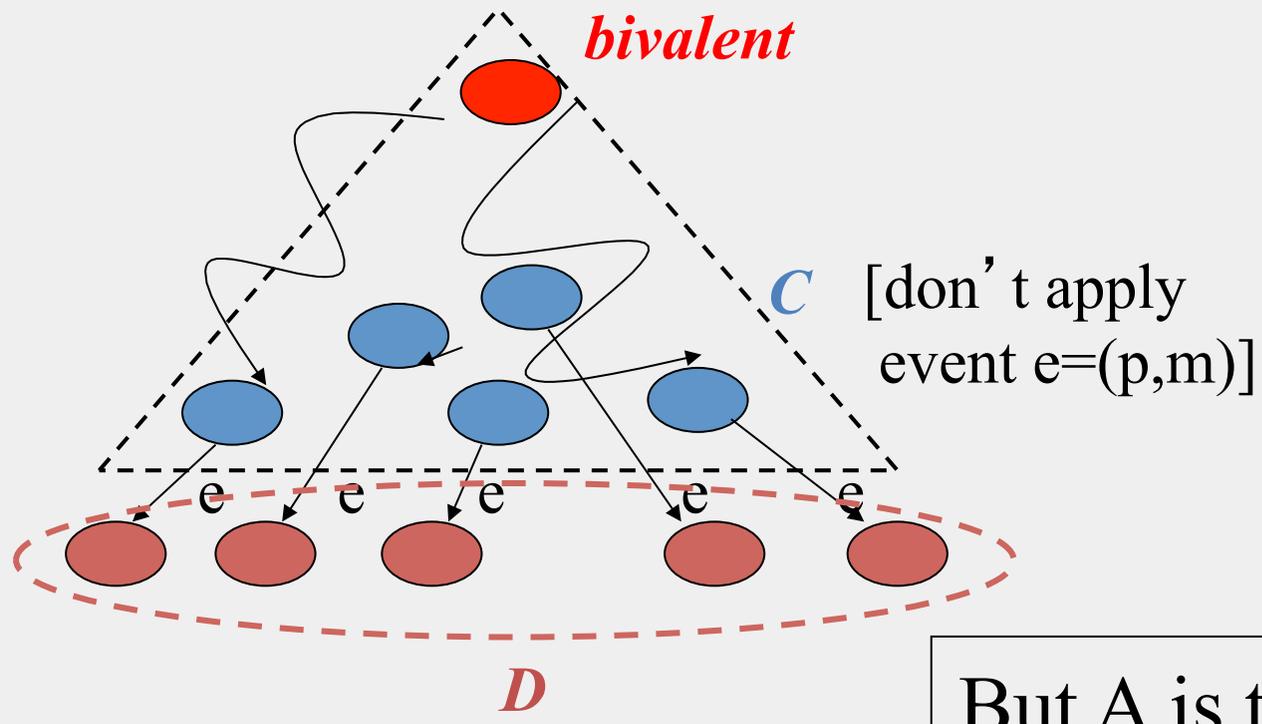
- Case I:  $p'$  is not  $p$
- Case II:  $p'$  same as  $p$



sch. s



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



But A is then bivalent!

# LEMMA 3

Starting from a bivalent config., there is always another bivalent config. that is reachable

# PUTTING IT ALL TOGETHER

- Lemma 2: There exists an initial configuration that is bivalent
- Lemma 3: Starting from a bivalent config., there is always another bivalent config. that is reachable
- Theorem (Impossibility of Consensus): **There is always a run of events in an asynchronous distributed system such that the group of processes never reach consensus (i.e., stays bivalent all the time)**

# 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 even one (adversarial) crash-stop process failure, there are always sequences of events for the system to decide any which way
    - Holds true regardless of whatever algorithm you choose!
  - FLP impossibility proof
- One of the most fundamental results in distributed systems



# ANNOUNCEMENTS: MIDTERM

- Midterm next Tuesday
- In-class
- Closed-book, closed-notes.
- **Calculators ok.**
- **No cellphones or other devices, etc.**
- Bring pen/pencil
- Multiple choice (short), fill in the blanks (long), design question (longer)
- Sample midterm problems on website
- Also look at HWs, videos, slides

