# Review

- In your own words, define the following properties for multicast:
- FIFO ordering
- Causal ordering
- Total ordering

# ECEN 757: Consensus

Chapter 15

# System Model

- Communication is reliable
  - Asynchronous: Delay is not bounded
  - Synchronous: Delay is bounded
- Processes can fail
  - Crash failure: Failed processes crash
  - Byzantine failure: Failed processes can have arbitrary behavior
- At most f out of N processes are faulty. All other processes are correct

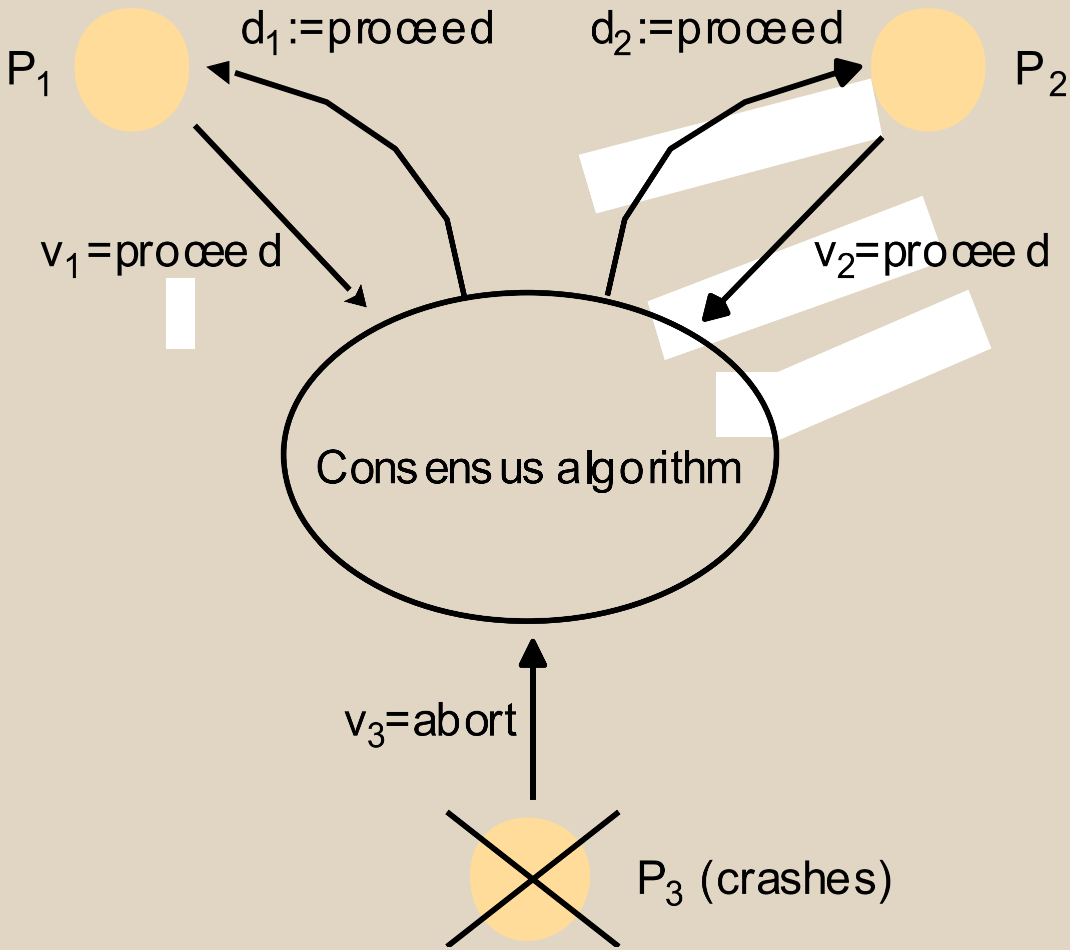
# Problem Definition

- At the beginning, each process Pi begins in the "undecided" state, and proposes a single value Vi
- Processes communicate with each other
- Eventually, Pi sets a decision variable Di, and enters the "decided" state
- What are the required properties?

# Requirement of Consensus

- Termination: Eventually each correct process sets its decision variable
- Agreement: The decision value of all correct processes is the same: if Pi and Pj are correct and have entered the “decided” state, then Di=Dj
- Integrity: If the correct processes all proposed the same value, then any correct process in the “decided” state has chosen that value

# Example



$P_1$
$d_1$:=proceed
$d_2$:=proceed
$P_2$
$v_1$=proceed
$v_2$=proceed
Consensus algorithm
$v_3$=abort
$P_3$ (crashes)

# Let’s Try to Solve Consensus!

- System Model 1:
- Synchronous system
- All processes are correct

# Solution

- When the algorithm starts: Everyone broadcasts its proposed value
- Each process waits until it obtains all values from others
- Processes choose the majority of the proposed values as the decision variable
  - Other criteria, such as max and min, also work

# Requirement of Consensus

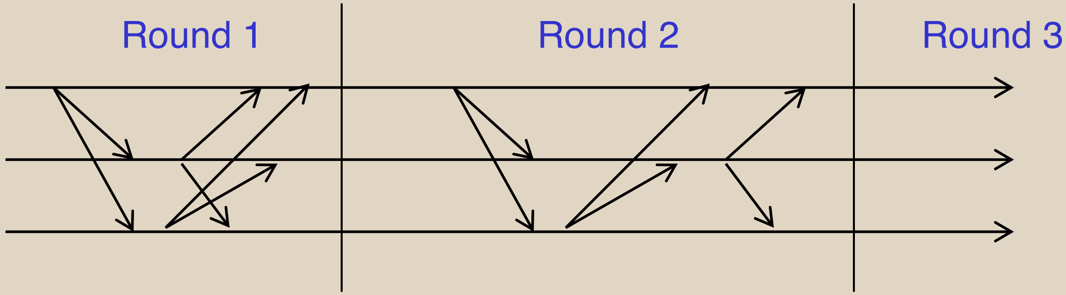
- Termination: Eventually each correct process sets its decision variable
- Agreement: The decision value of all correct processes is the same: if Pi and Pj are correct and have entered the “decided” state, then Di=Dj
- Integrity: If the correct processes all proposed the same value, then any correct process in the “decided” state has chosen that value

# Let’s Try to Solve Consensus!

- System Model 2:
- Synchronous system
- Processes can only fail by crashing
- At most f out of N processes can fail

# 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 >> 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 $r$.



Algorithm for process $p_i \in g$; algorithm proceeds in $f + 1$ rounds

*On initialization*

$Values_i^1 := \{v_i\}$; $Values_i^0 = \{\}$;

*In round* $r$ $(1 \le r \le f + 1)$

$B\text{-}multicast(g,\ Values_i^r - Values_i^{r-1})$; // Send only values that have not been sent

$Values_i^{r+1} := Values_i^r$;

*while* (in round $r$)

{

*On* $B\text{-}deliver(V_j)$ *from some* $p_j$

$Values_i^{r+1} := Values_i^{r+1} \cup V_j$;

}

*After* $(f + 1)$ *rounds*

Assign $d_i = minimum(Values_i^{f+1})$;

# 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.
- Does the system have integrity?

# Let’s Try to Solve Consensus!

- System Model 3:
- Synchronous system
- Byzantine failure
- At most f out of N processes can fail

# The Byzantine General Problem

- 3 or more generals are to agree to attack or to retreat
- One general, the commander, issues the order
- The others, lieutenants, need to decide whether to attack or to retreat
- Some generals, including the commander, can be "treacherous"
- Commander: he proposes attacking to one general, and retreating to another
- Lieutenant: He tells one of his peer that the commander told him to attack, and another that they are to retreat

# Requirements

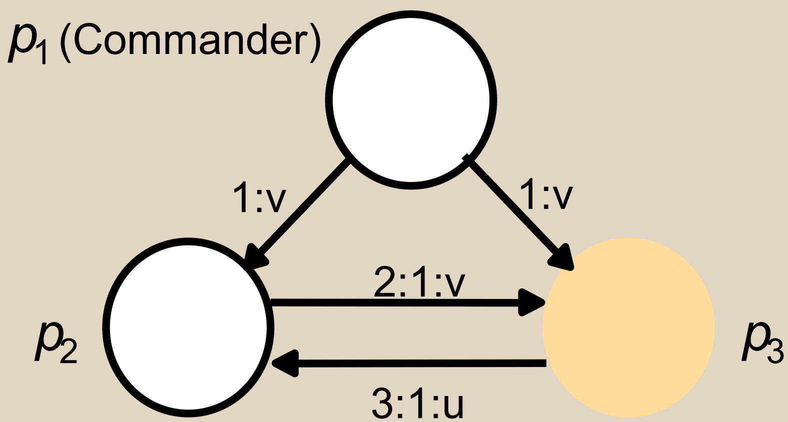
- Termination: Eventually each correct process sets its decision variable
- Agreement: The decision value of all correct processes is the same
- Integrity: If the commander is correct, then all correct processes decide on the value that the commander proposed

# The Byzantine General Problem in a Synchronous System

- There is no solution if and only if $N \leq 3f$

# Impossibility with 3 Processes

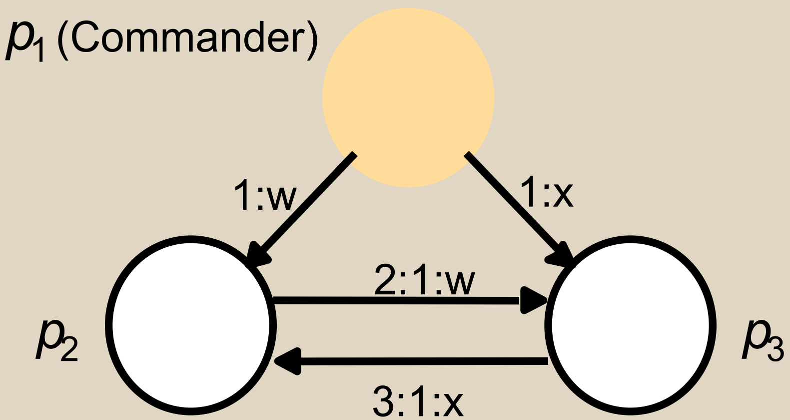
- Consider the following two scenarios:
- In this scenario, p2 should choose v, i.e., it should obey the commander




Faulty processes are shown coloured

# Impossibility with 3 Processes

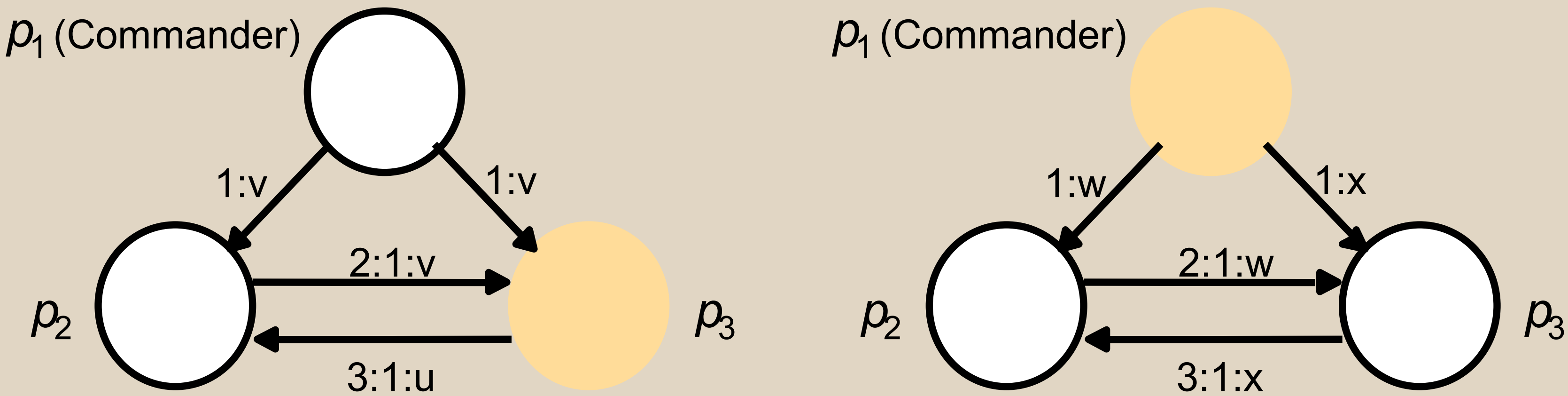
- Consider the following two scenarios:
- In this scenario, one of p2 and p3 need to disobey the commander. Otherwise, agreement cannot be met




Faulty processes are shown coloured

# Impossibility with 3 Processes

- From p2’s point of view, there is no way it can determine which scenario it is in




Faulty processes are shown coloured

# Impossibility with $N \leq 3f$

- Divide processes into 3 groups, each with $n_1, n_2, n_3$, processes
- Make sure $n_i \leq f$
- Processes in group i play the role of pi in the 3-processes example
- It is then impossible to achieve consensus

# Consensus with $N = 4, f = 1$

- Each lieutenant receives 3 messages: one from the commander, and the other from the two other lieutenants
- Decision = majority

# 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:

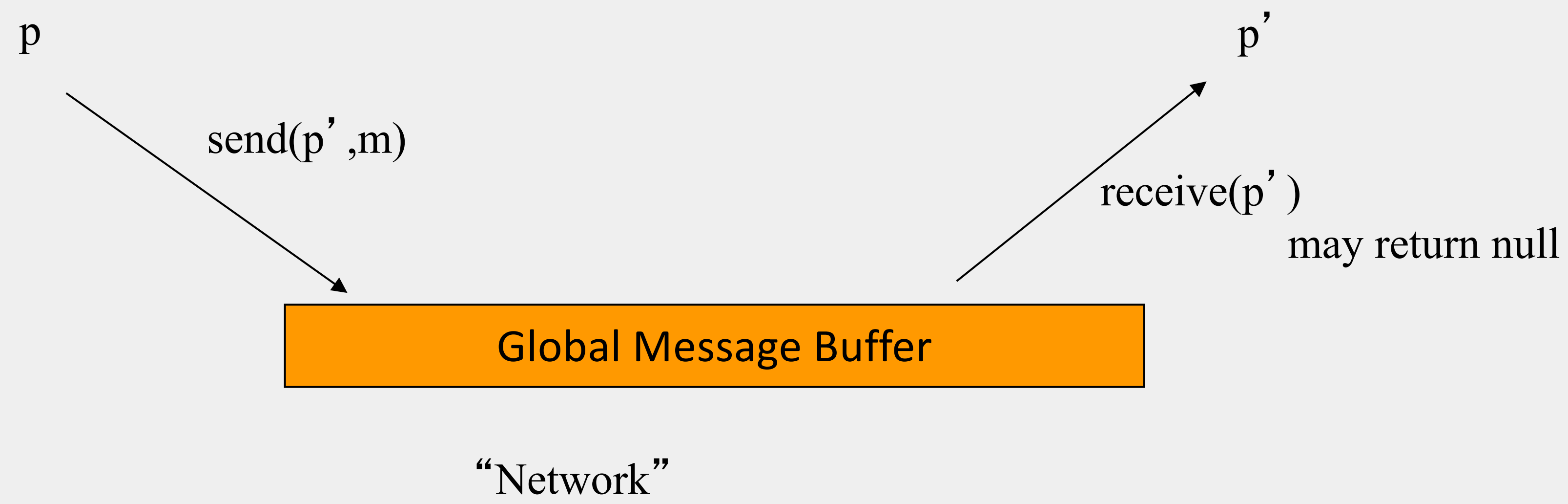
- Each process p has a state
  - program counter, registers, stack, local variables
  - input register xp : initially either 0 or 1
  - output register yp : initially b (undecided)
- Consensus Problem: design a protocol so that either
  - all processes set their output variables to 0 (all-0's)
  - Or 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

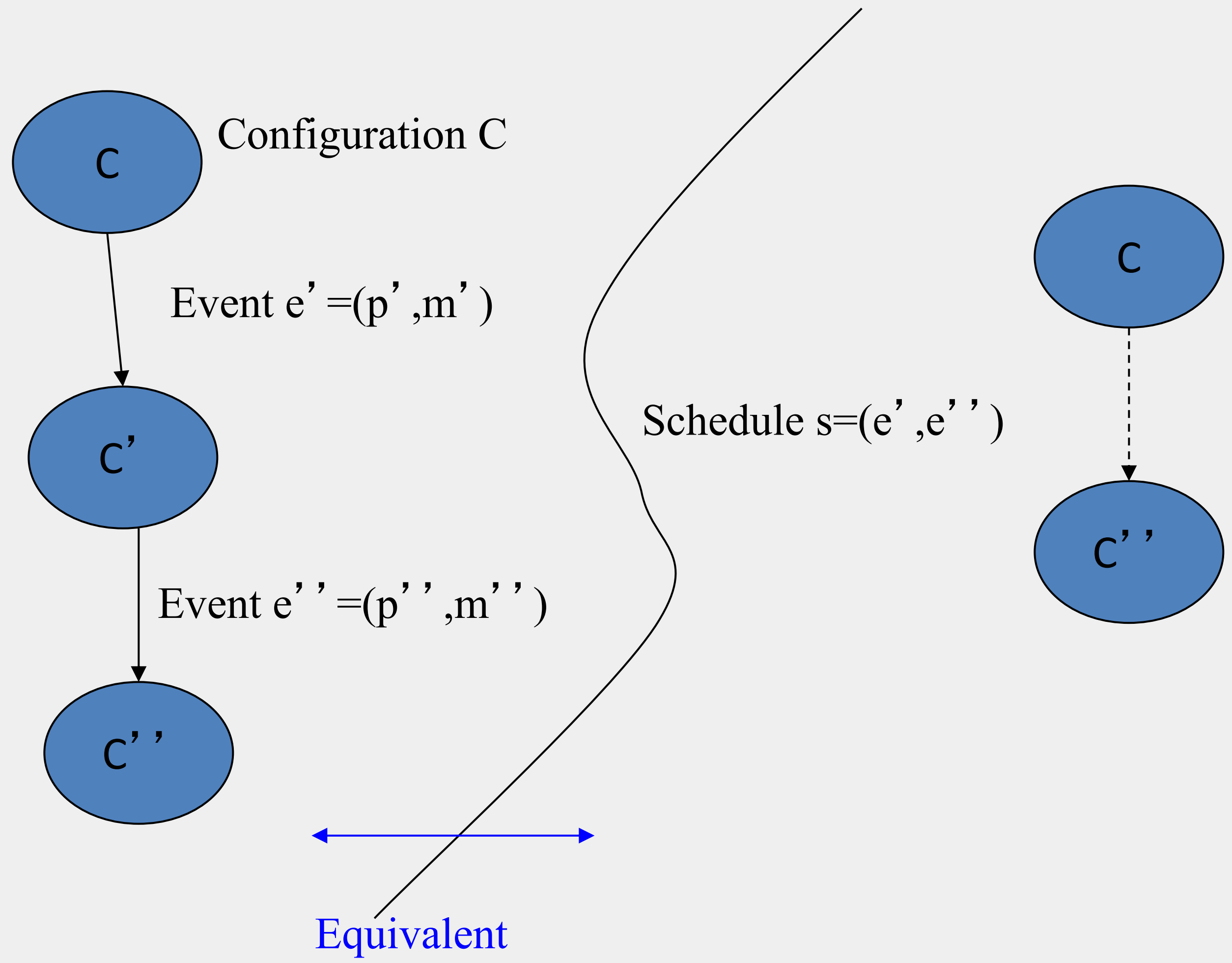
1. more restrictive system model, and
2. easier problem
    - Why is this is ok?

# Network

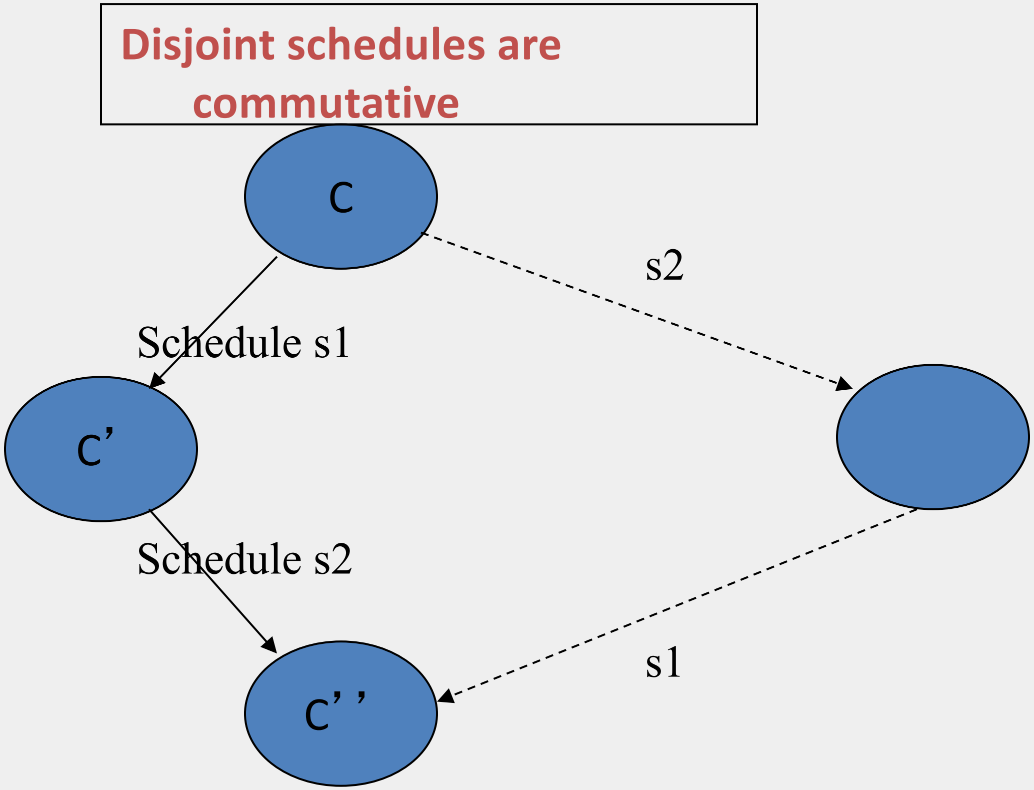

p
p'
send(p',m)
receive(p')
may return null
Global Message Buffer
"Network"

# 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


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

# Lemma 1




s1 and s2 involve <u>disjoint</u> sets of <u>receiving</u> processes, and are <u>each</u> applicable on C

# Easier Consensus Problem

Easier Consensus Problem:

some process eventually sets yp 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

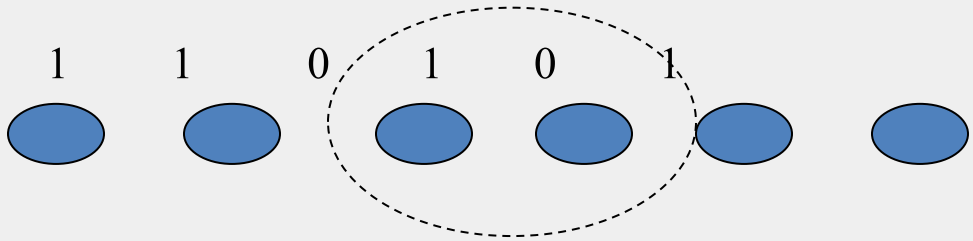
1. There exists an initial configuration that is bivalent

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.

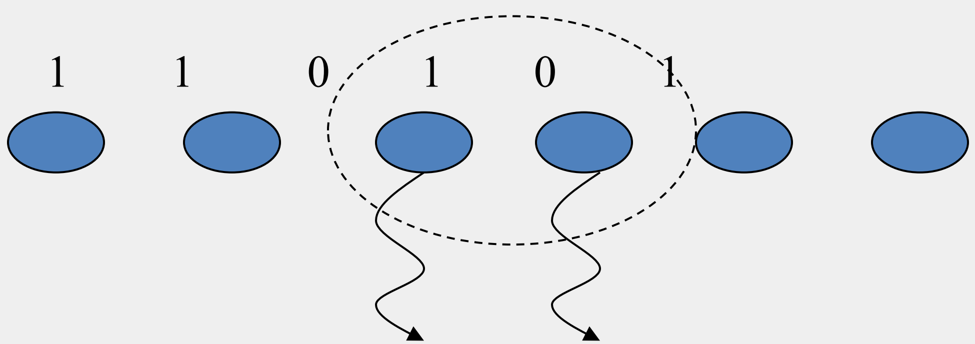



- There has to be some adjacent pair of 1-valent and 0-valent configs.

# 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 <u>bivalent</u> when there is such a failure

# What we'll show

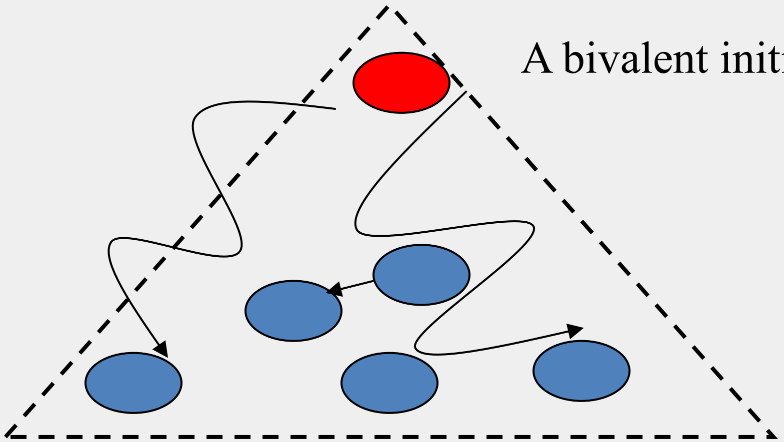
1. There exists an initial configuration that is bivalent

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

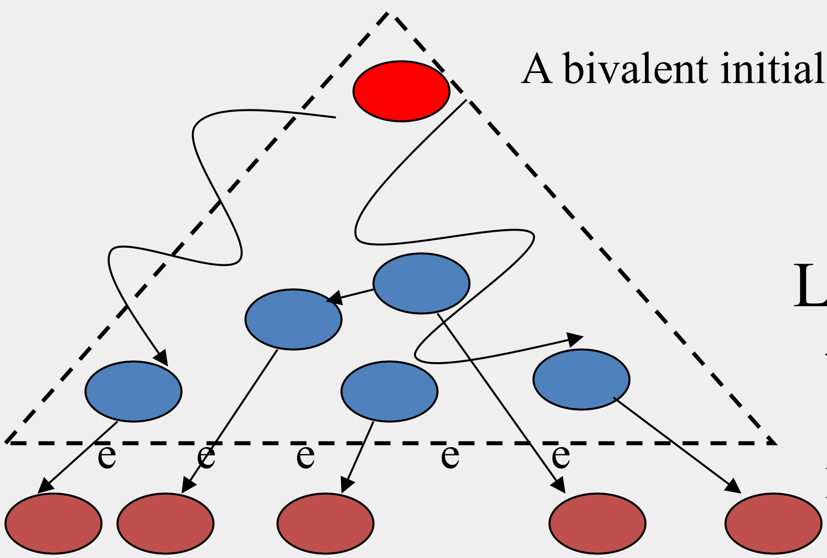


A bivalent initial config.

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



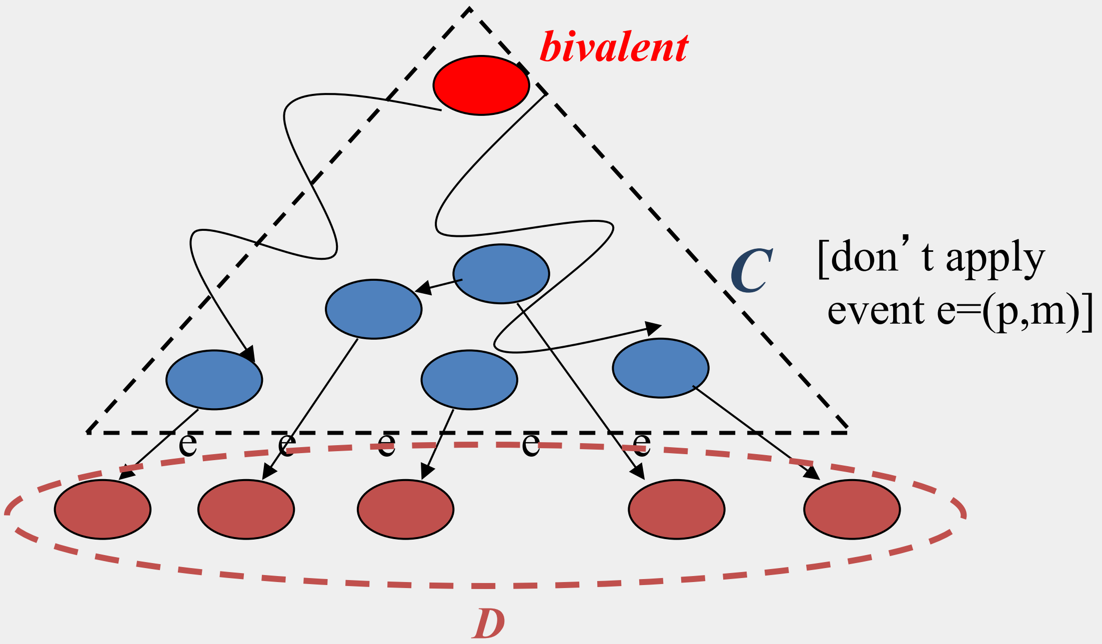

A bivalent initial config.

let e=(p,m) be some event applicable to the initial config.
(p,m) means p receives m.
The event can be arbitrarily delayed

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

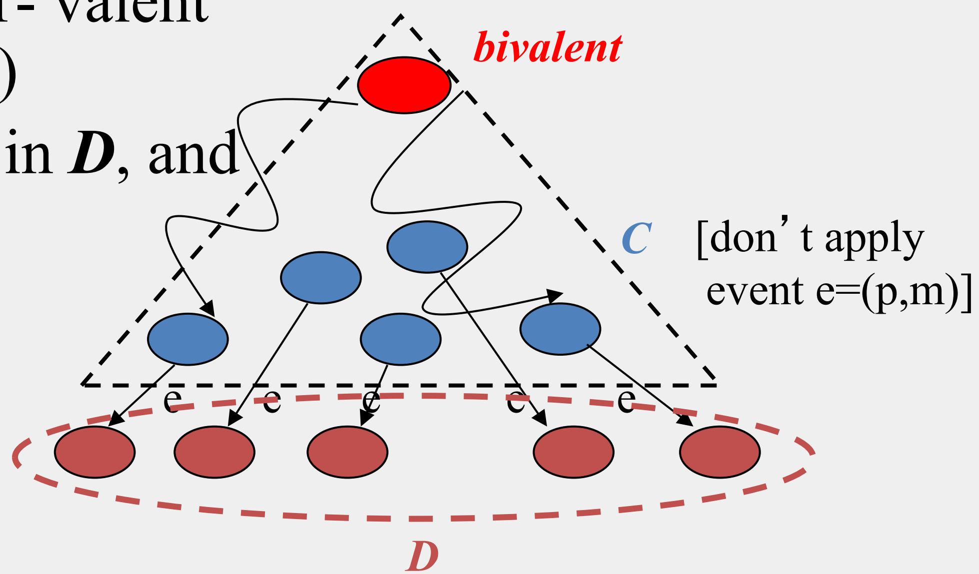


**Claim.** Set D contains a bivalent config.

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

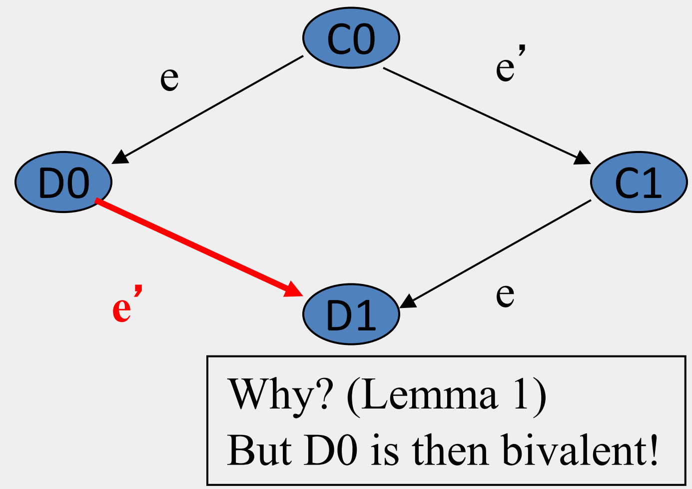
- There are states D0 and D1 in ***D***, and C0 and C1 in ***C*** such that
  - D0 is 0-valent, D1 is 1-valent
  - D0=C0 foll. by e=(p,m)
  - D1=C1 foll. by e=(p,m)
  - And C1 = C0 followed by some event e' =(p' ,m' )

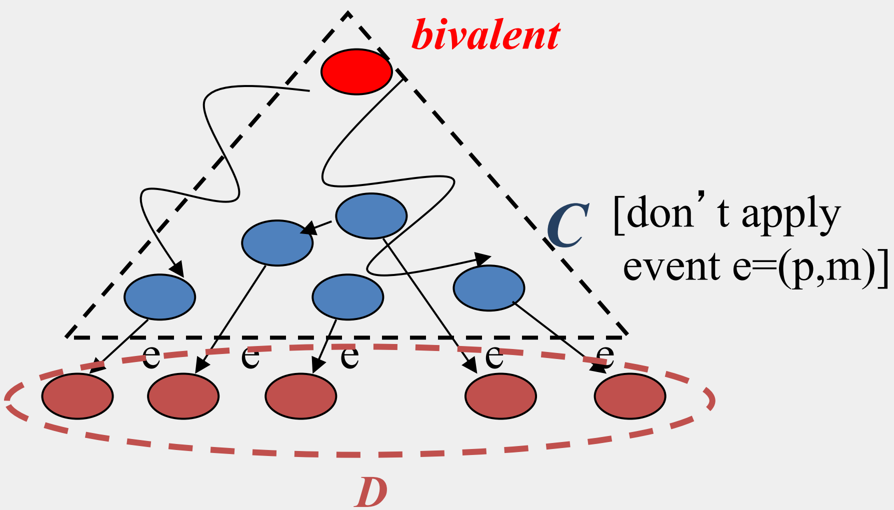
(why?)



**Proof.** (contd.)

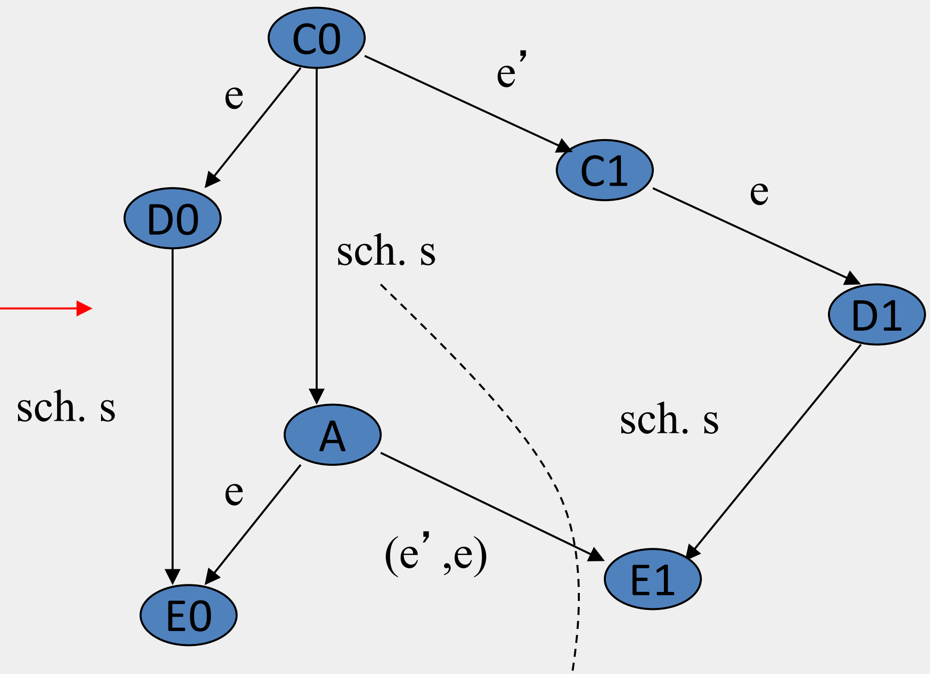
- Case I: p' is not p
- Case II: p' same as p

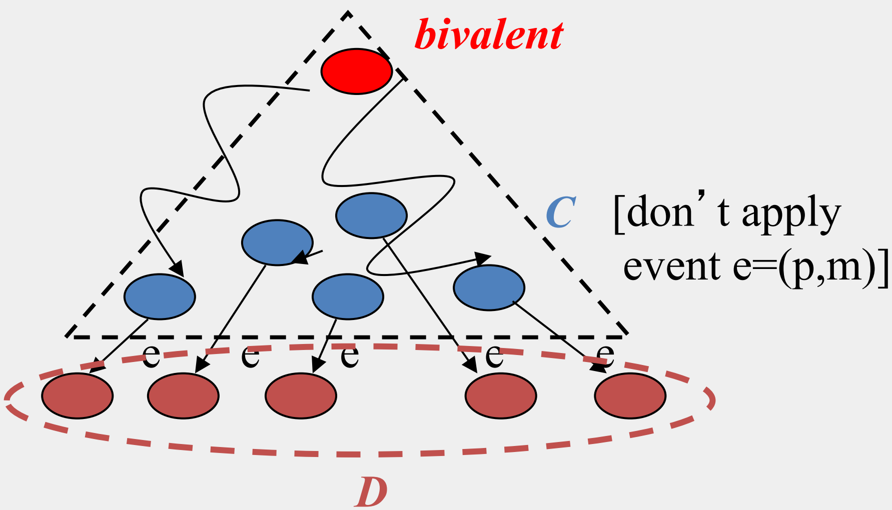


**Proof.** (contd.)

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




But A is then bivalent!

sch. s
- finite
- **deciding run** from C0
- Deciding run = some process reach the decided state during the run
- *p takes no steps*
- Sch. s exists, or the system cannot handle p's failure

# 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)