Consensus

CS425 /ECE428 – DISTRIBUTED SYSTEMS – FALL 2021

Material derived from slides by I. Gupta, M. Harandi, J. Hou, S. Mitra, K. Nahrstedt, N. Vaidya

Consensus: Example

Proposal: move midterm date to another day

Consensus needed

- All students must be OK with new date (input)
- Everyone must know the final decision (agreement)

What is Consensus?

N processes

Each process p has

- input variable x_p : initially either 0 or 1
- output variable y_p : initially b (b=undecided) – can be changed only once

Consensus problem: design a protocol so that either

- all non-faulty processes set their output variables to 0
- *Or* all non-faulty processes set their output variables to 1
- There is at least one initial state that leads to each outcomes 1 and 2 above

System Model!

Processes fail only by crash-stopping

Synchronous system: bounds on:

- Message delays
- Max time for each process step

Asynchronous system: no such bounds

Consensus, try 1

1. Each process p_i multicasts it's input x_i to all other processes
2. Upon receiving x_j from all processes p_j, set $y_i = f(x_1, \ldots, x_n)$
   - E.g., $y_i = \min(x_1, \ldots, x_n)$

Synchronous Consensus (try 2)

1. Each process p_i multicasts it's input x_i to all other processes
2. If a process does not reply with a timeout
3. Upon receiving x_j from all processes p_j, set $y_i = f(x_1, \ldots, x_n)$ (omitting any processes that timed out)
   - E.g., $y_i = \min(x_1, \ldots, x_n)$

Consensus Problem

System of N processes (P_1, P_2, ….., P_n)

Each process P_i:

- begins in an *undecided* state.
- proposes value v_i.
- at some point during the run of a consensus algorithm, sets a decision variable d_i and enters the *decided* state.

Required Properties

Termination: Eventually each process sets its decision variable.

Agreement: The decision value of all correct processes is the same.

- If P_i and P_j are correct and have entered the *decided* state, then $d_i = d_j$.

Integrity (non-triviality): If all the correct processes proposed the same value, then any correct process in the *decided* state has chosen that value.

or There is some initial state and execution that leads to deciding 0 and to deciding 1

How do we agree on a value?

Ring-based leader election

- Send proposed value along with *elected* message.
- Turnaround time: 3NT worst case and 2NT best case (without failures).
  - T is the time taken to transmit a message on a channel.
- O(Nft) if up to f processes fail during the election run.

Bully algorithm

- Send proposed value along with the *coordinator* message.
- Turnaround time: 4T in the worst case without failures.
- More than 2fT if up to f processes fail during the election run.

What's the best we can do?

Consider the simplest algorithm

Let's assume the system is synchronous.

Use a simple B-multicast:

- All processes B-multicast their proposed value to all other processes.
- Upon receiving all proposed values, pick the minimum.

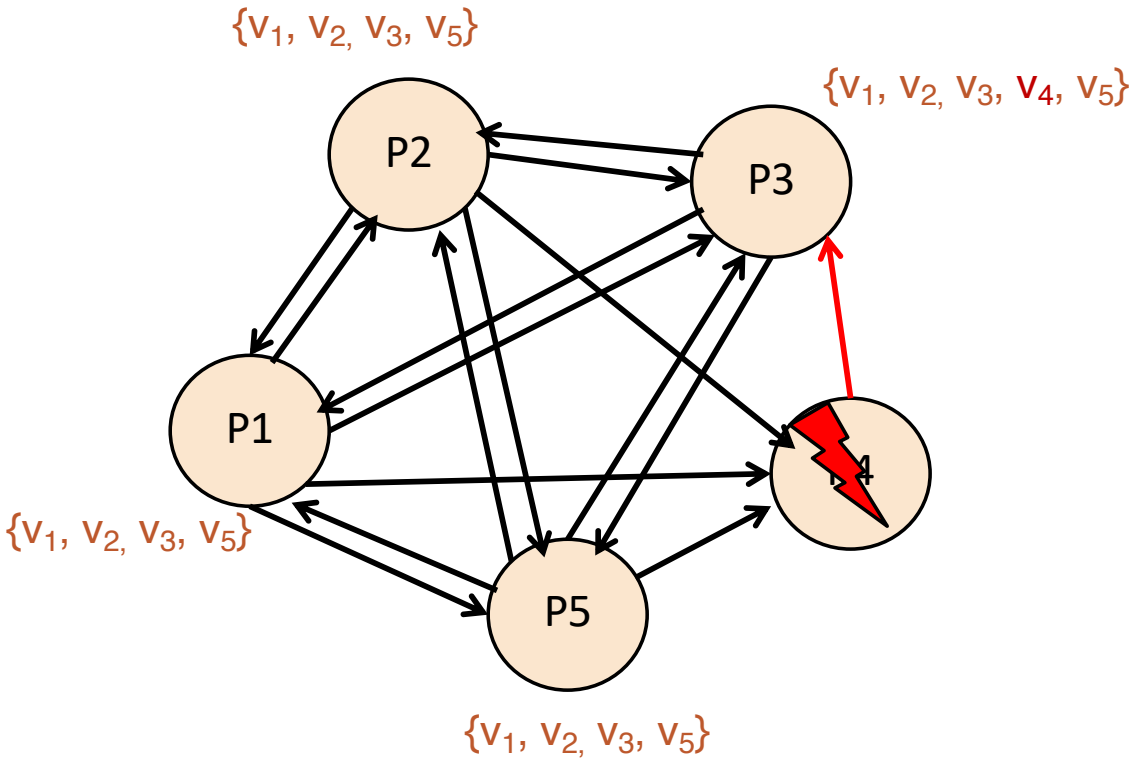
Time taken under no failures?

- One message transmission time (T)

What can go wrong?

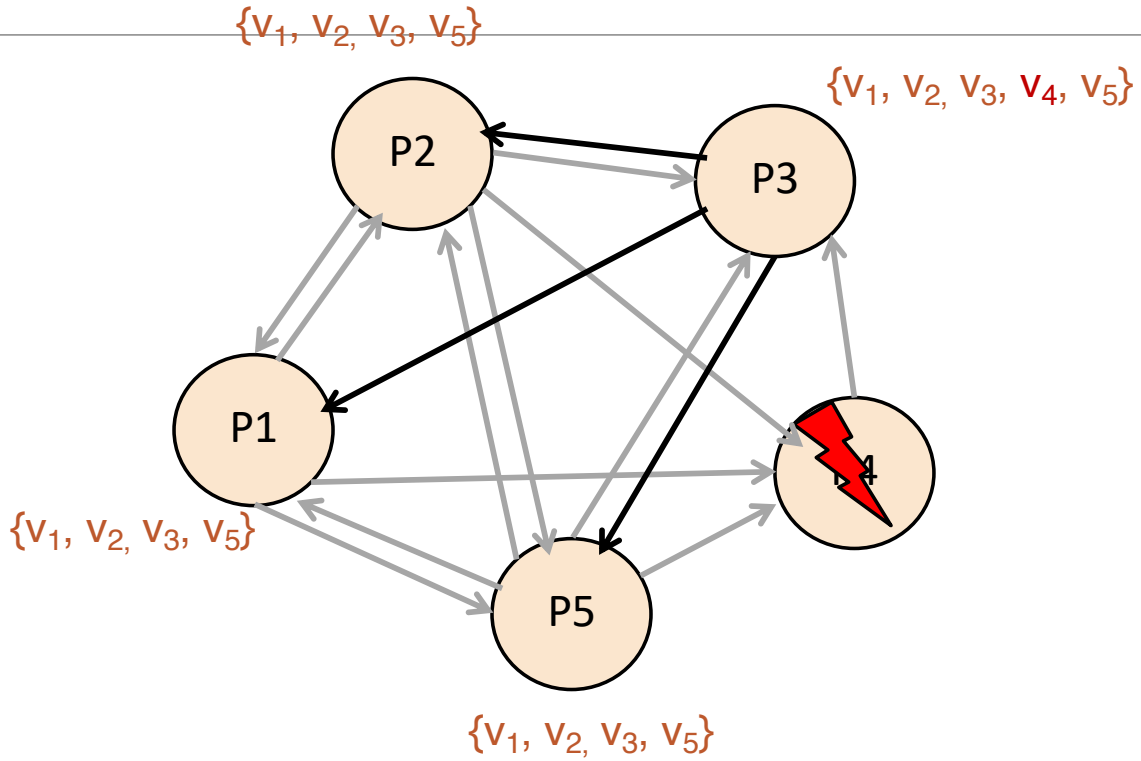
- If we consider process failures, is a simple B-multicast enough?

B-multicast is not enough for this



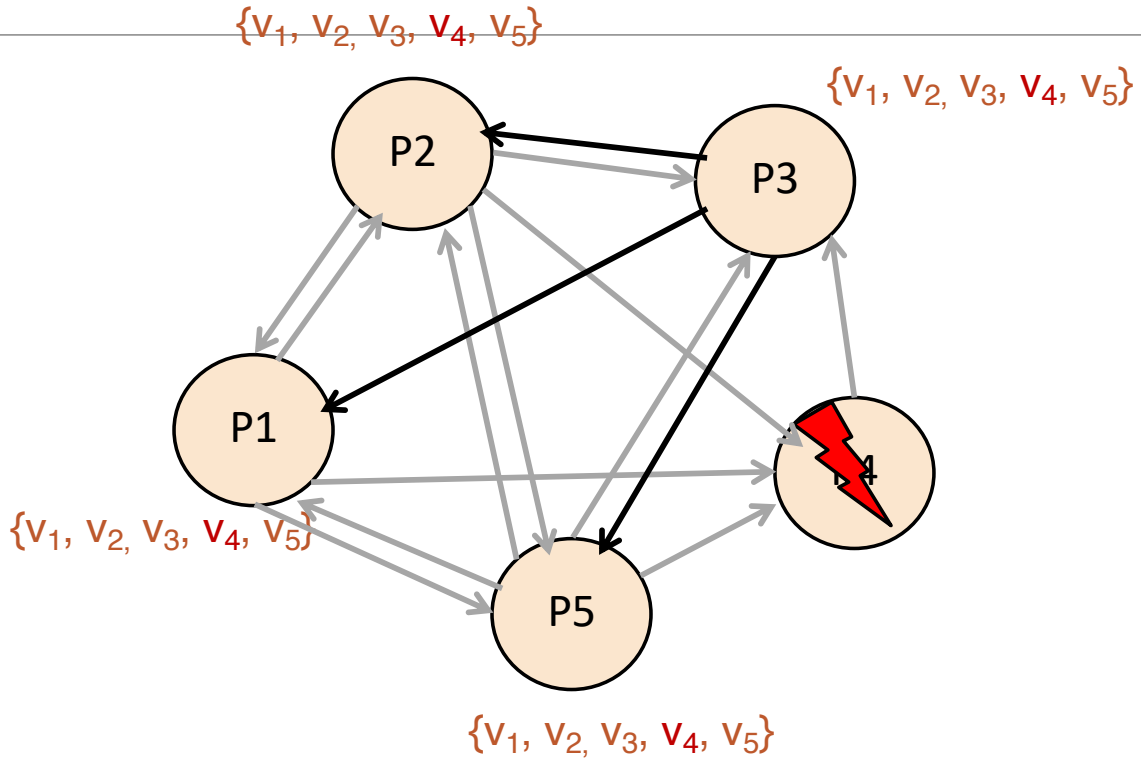
Need R-multicast

B-multicast is not enough for this



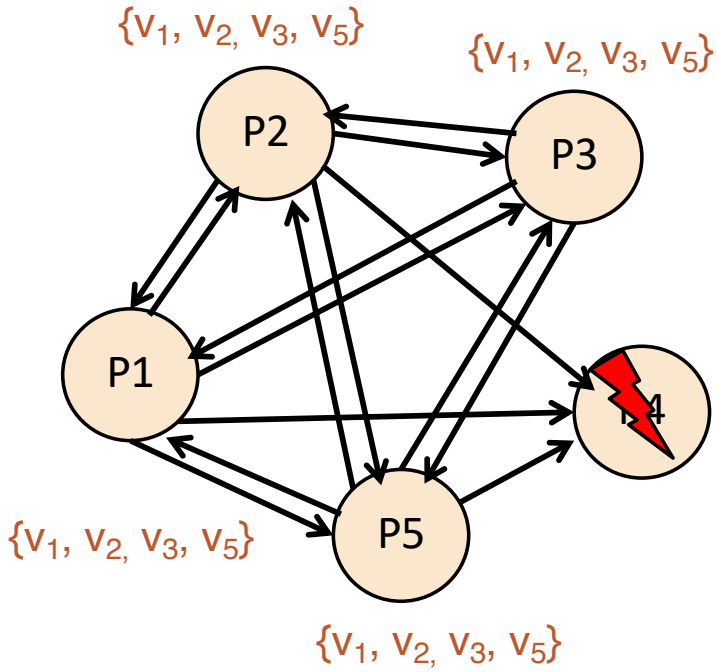
Need R-multicast

B-multicast is not enough for this



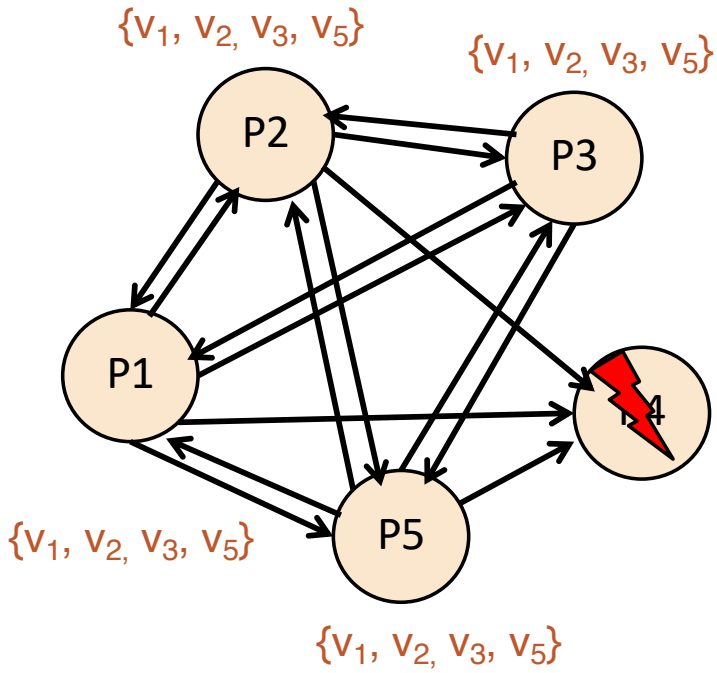
Need R-multicast

Handling failures



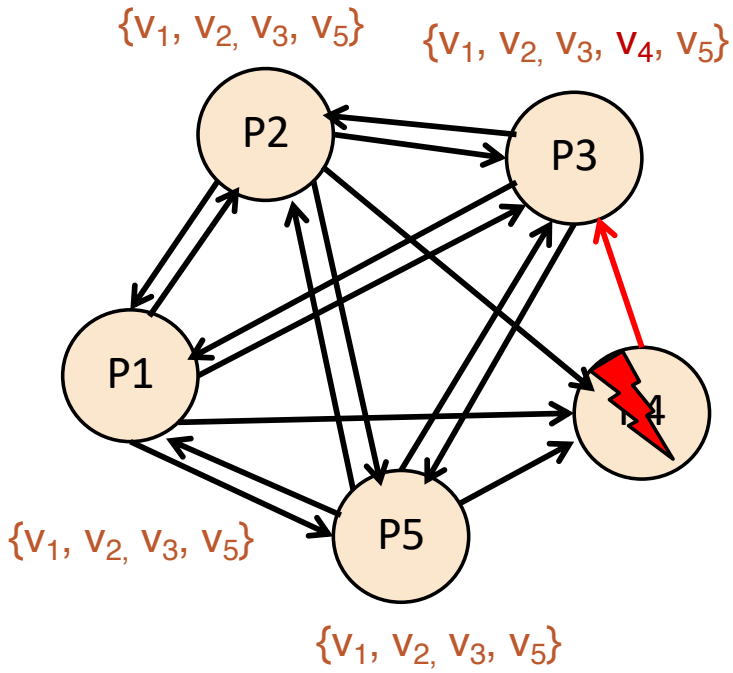
- P4 fails before sending v_4 to anyone.
- What should other processes do?
- Detect failure. *Timeout!*
- Assume proposals are sent at time ‘s’.
- Worst-case skew is ϵ.
- Maximum message transfer time (including local processing) is T.
- What should the timeout value be?

Handling failures



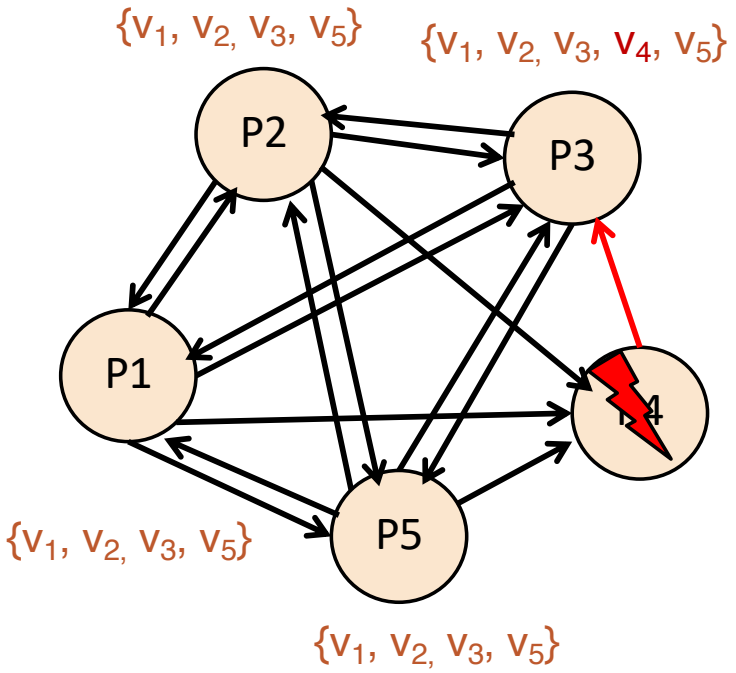
- Assume proposals are sent at time ‘s’.
- Worst-case skew is ϵ.
- Maximum message transfer time (including local processing) is T.
- What should the timeout value be?
- Option 1: $\epsilon + T$
  - Pi waits for ($\epsilon + T$) time units after sending its proposal at time ‘s’.
  - Any other process must have sent proposed value before s + ϵ.
  - The proposed value should have reached Pi by (s + ϵ + T).
  - *Will this work?*

Handling failures



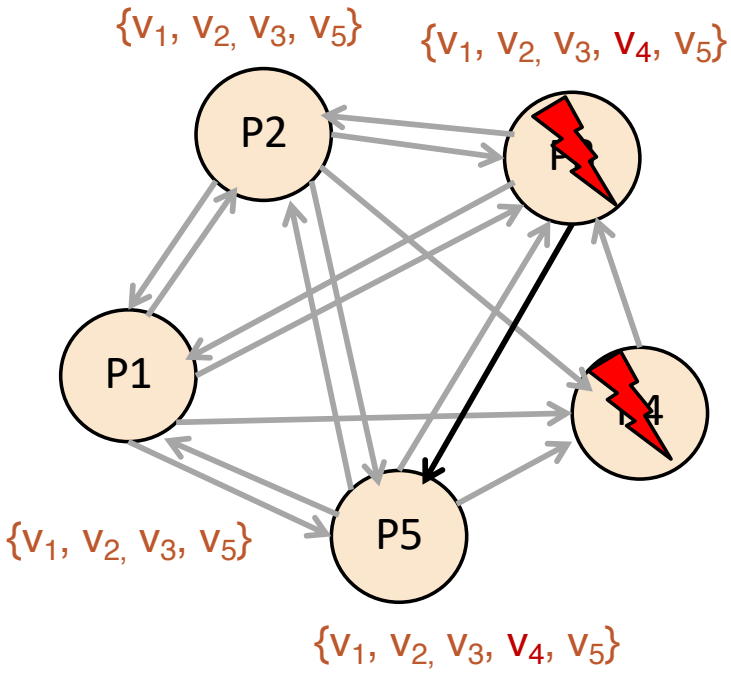
- Assume proposals are sent at time ‘s’.
- Worst-case skew is ϵ.
- Maximum message transfer time (including local processing) is T.
- What should the timeout value be?
- Option 1: $\epsilon + T$
  - Pi waits for ($\epsilon + T$) time units after sending its proposal at time ‘s’.
  - Any other process must have sent proposed value before s + ϵ.
  - The proposed value should have reached Pi by (s + ϵ + T).
  - *Will this work?*

Handling failures

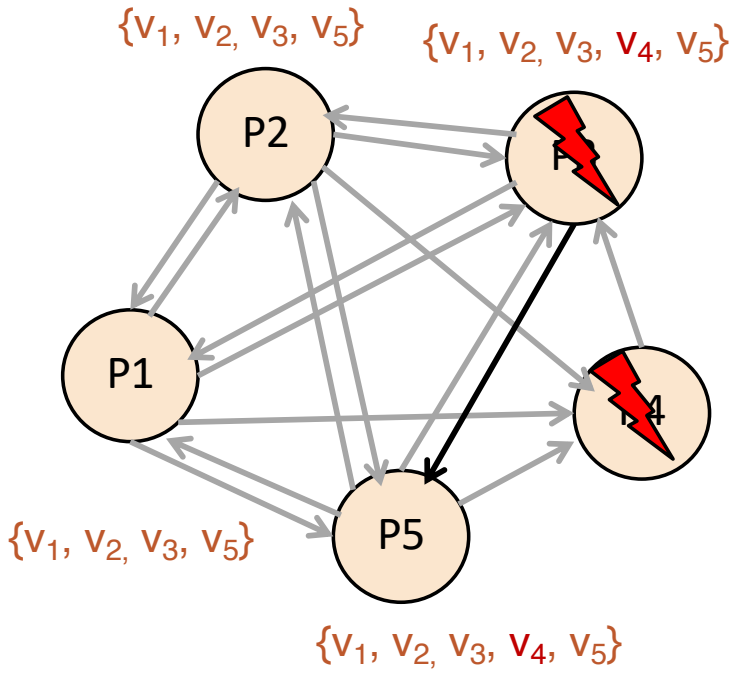


- Assume proposals are sent at time ‘s’.
- Worst-case skew is ϵ.
- Maximum message transfer time (including local processing) is T.
- What should the timeout value be?
- How about ϵ + 2*T?
  - *Will this work?*

Handling failures

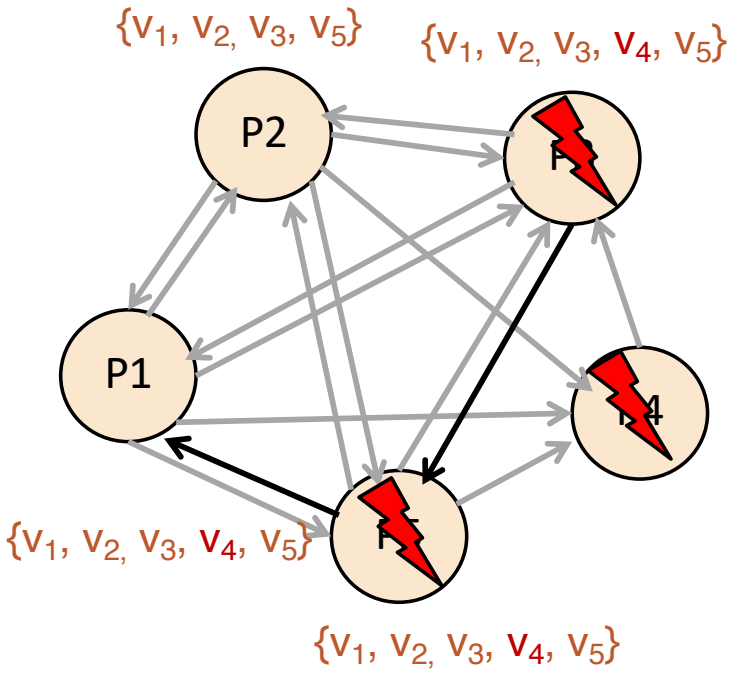


Handling failures



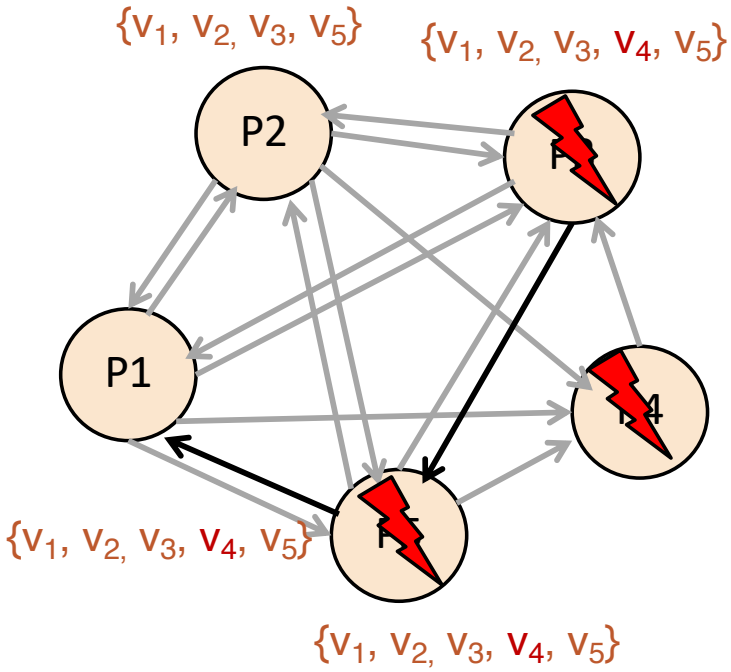
- Assume proposals are sent at time ‘s’.
- Worst-case skew is ϵ.
- Maximum message transfer time (including local processing) is T.
- What should the timeout value be?
- How about ϵ + 3*T?
  - *Will this work?*

Handling failures



- Assume proposals are sent at time ‘s’.
- Worst-case skew is ϵ.
- Maximum message transfer time (including local processing) is T.
- What should the timeout value be?
- How about ϵ + 3*T?
  - *Will this work?*

Handling failures



- Assume proposals are sent at time ‘s’.
- Worst-case skew is ϵ.
- Maximum message transfer time (including local processing) is T.
- What should the timeout value be?
- Timeout = ϵ + (f+1)*T for up to f failed process.

Also holds for R-multicast from a single sender.

Round-based algorithm

- For a system with at most f processes crashing
  - All processes are *synchronized* and operate in "rounds" of time.
    - One round of time is equivalent to $\epsilon + T$ units.
    - At each process, the i^{th} round
      - starts at local time $s + (i-1)*(\epsilon + T)$
      - ends at local time $s + i*(\epsilon + T)$
    - The start or end time of a round in two different processes differs by at most ϵ.
  - The algorithm proceeds in f+1 rounds.
  - Assume communication channels are reliable.

Consensus in Synchronous Systems

```
import time

TIME_BOUND = 100 # enough time to
# receive all msgs in a round

def consensus(input,max_fail):
  """
   `input`: this processes's input
   `max_fail`: number of process
         failures to tolerate
  """
values = { input } # set of values
# currently known to this process

for round in range(max_fail+1):
  round_start = time.time()
  # send values to all other procs
  multicast(values)
  while time.time() < round_start+\
            TIME_BOUND:
    for message in get_messages():
      # add received values to ours
      values = values | message

return min(values) # or any
# deterministic function over the
# value set
```

Proof

Lemma 1: after a round with no failures, all non-faulty processes have the same values set

- **Proof**: if there are no failures, B-multicast is reliable
- All processes receiver the same messages

Main proof:

- There are more rounds than expected failures
- Therefore at least one round has no failures
- After this round all non-crashed processes have the same value set
- Value set will remain the same in future rounds

Consensus in an Asynchronous System

Messages have arbitrary delay, processes arbitrarily slow

Impossible to achieve!

- even a single failed is enough to avoid the system from reaching agreement!
- a slow process indistinguishable from a crashed process

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 a 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

- all non-faulty processes set their output variables to 0
- Or non-faulty all processes set their output variables to 1
- (No trivial solutions allowed)

States, Events, and Schedule

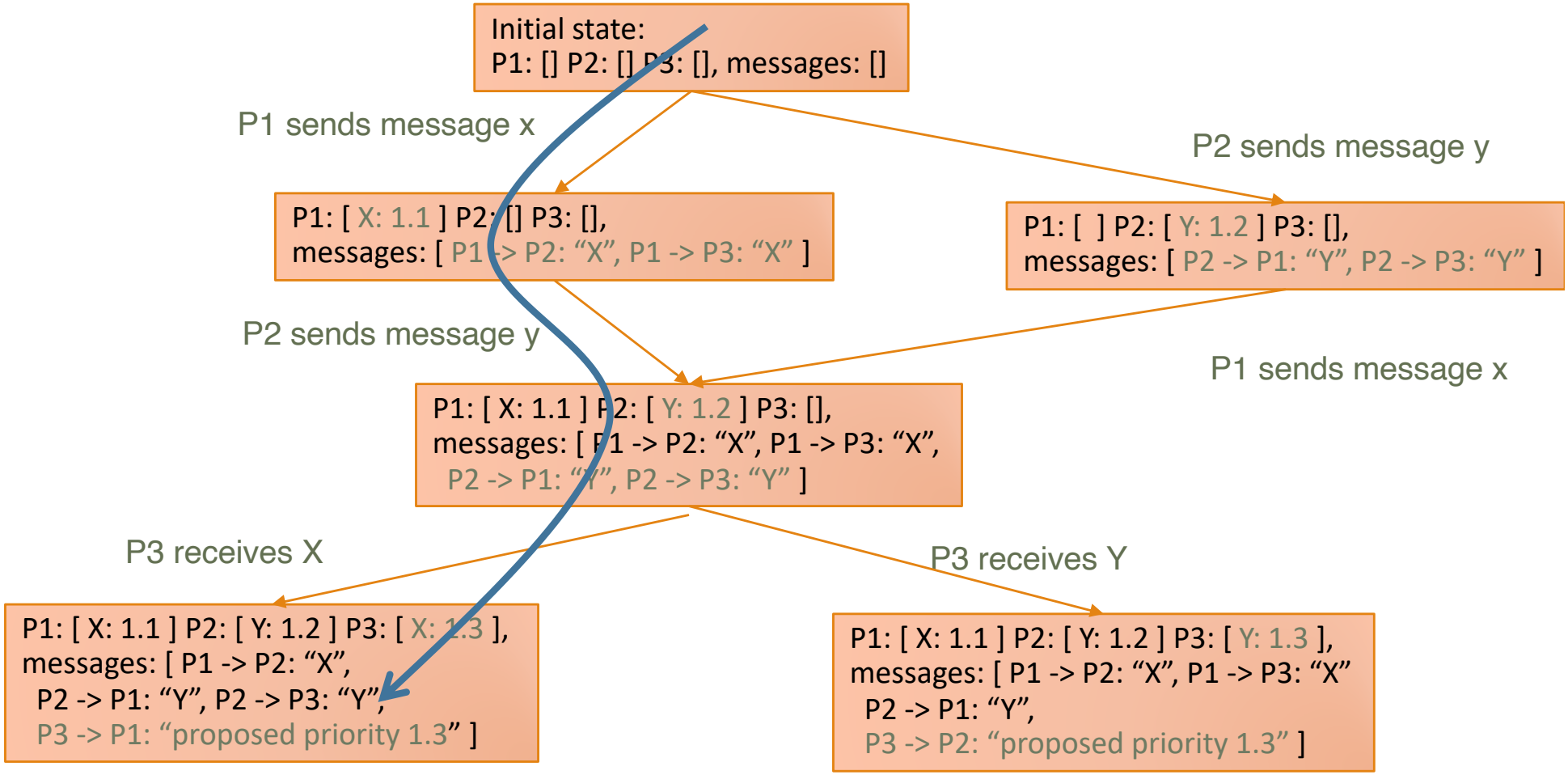
A *global state* consists of:

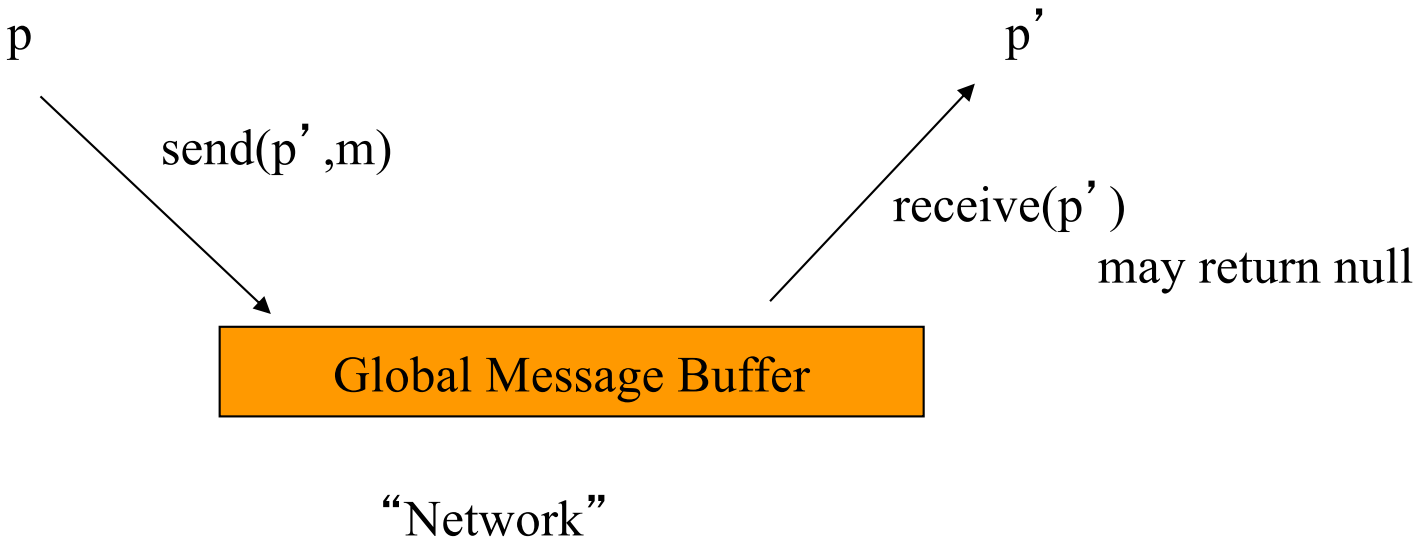
- The state of all processes
- The state of all communication channels (messages “in flight”)

An *event* involves transitions between global states

- Read one “in flight” message
- Do some processing
- Send one or more messages

A *schedule* is a sequence of events


Initial state:
P1: [] P2: [] P3: [], messages: []
P1 sends message x
P2 sends message y
P1: [X: 1.1] P2: [] P3: [],
messages: [P1 -> P2: "X", P1 -> P3: "X"]
P1: [ ] P2: [Y: 1.2] P3: [],
messages: [P2 -> P1: "Y", P2 -> P3: "Y"]
P2 sends message y
P1 sends message x
P1: [X: 1.1] P2: [Y: 1.2] P3: [],
messages: [P1 -> P2: "X", P1 -> P3: "X",
P2 -> P1: "Y", P2 -> P3: "Y"]
P3 receives X
P3 receives Y
P1: [X: 1.1] P2: [Y: 1.2] P3: [X: 1.3],
messages: [P1 -> P2: "X",
P2 -> P1: "Y", P2 -> P3: "Y",
P3 -> P1: "proposed priority 1.3"]
P1: [X: 1.1] P2: [Y: 1.2] P3: [Y: 1.3],
messages: [P1 -> P2: "X", P1 -> P3: "X"
P2 -> P1: "Y",
P3 -> P2: "proposed priority 1.3"]


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

Different Definition of "State"

State of a process

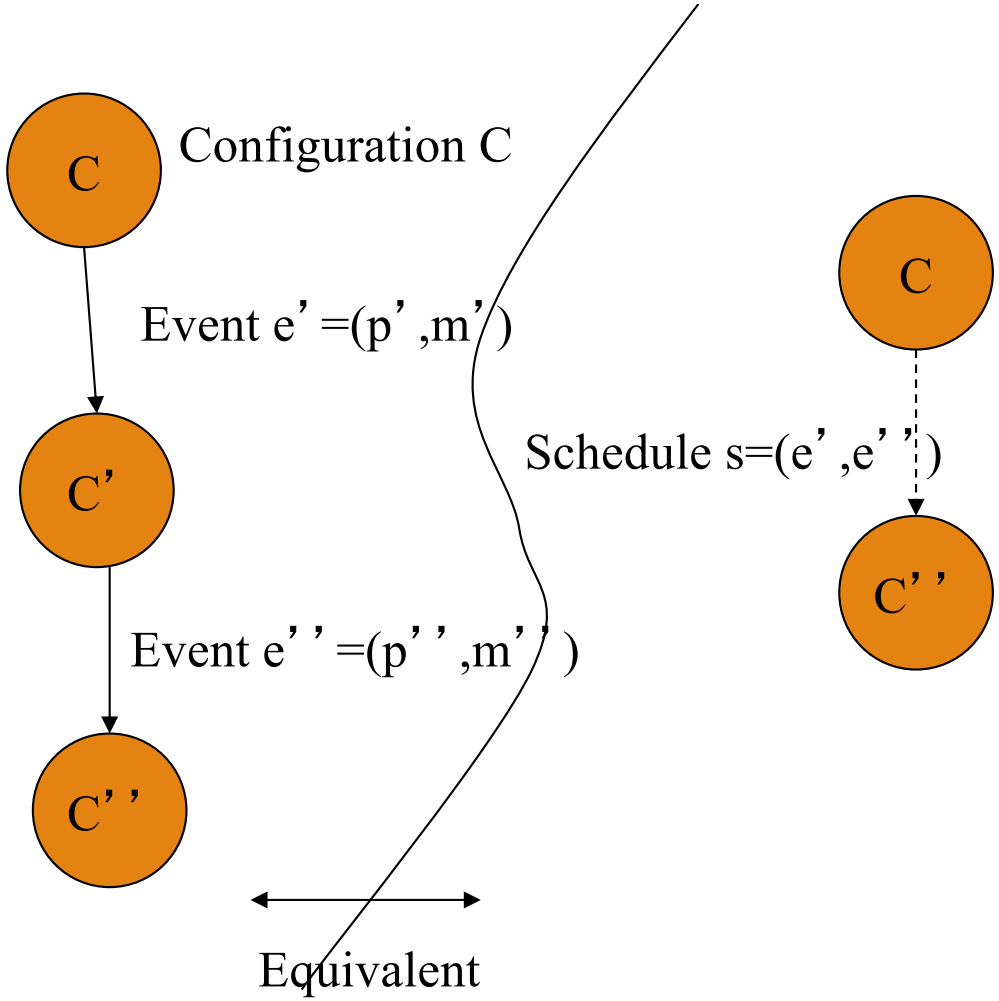
Configuration: = Global state. Collection of states, one per process; and state of the global buffer

Each Event consists atomically of three sub-steps:

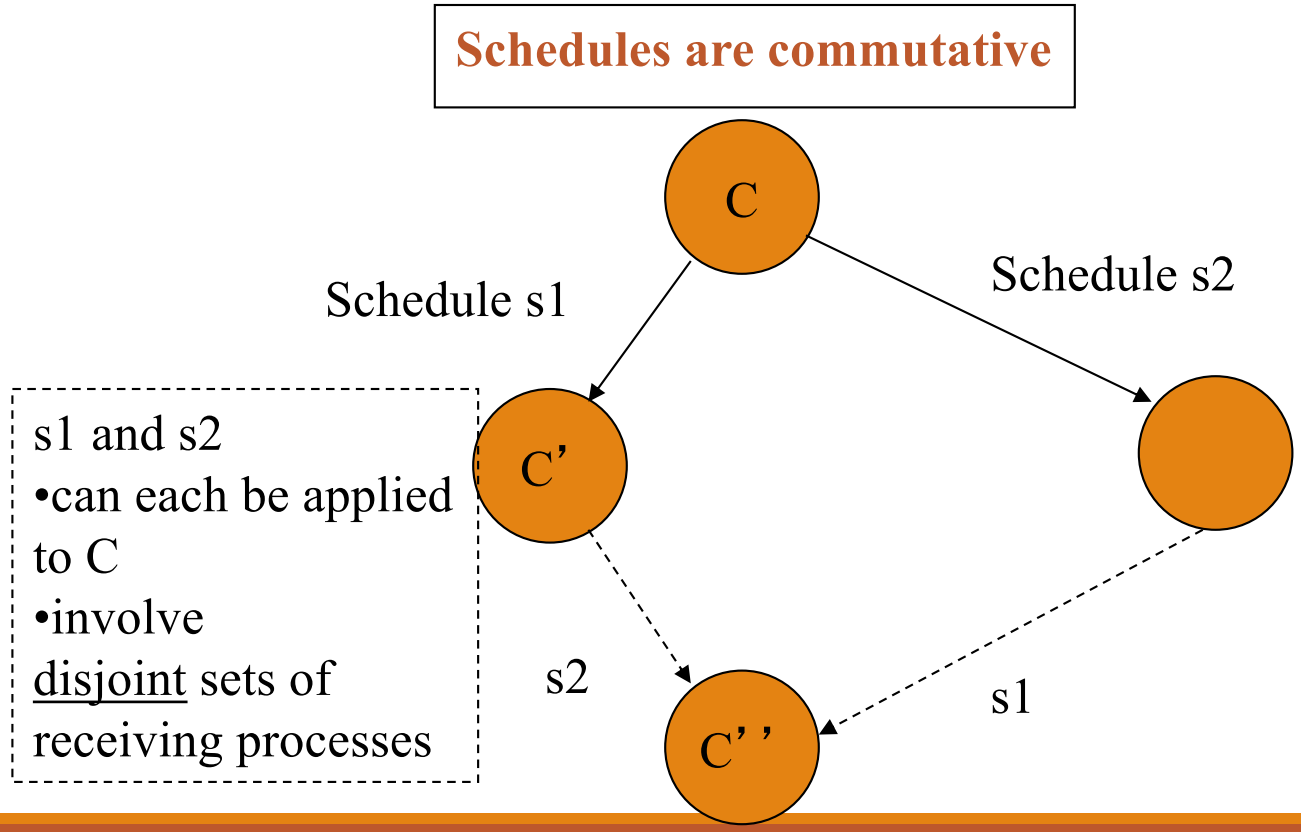
- receipt of a message by a process (say p), and
- processing of message, and
- sending out of all necessary messages by p (into the global message buffer)

Note: this event is different from the Lamport events

Schedule: sequence of events


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

Lemma 1

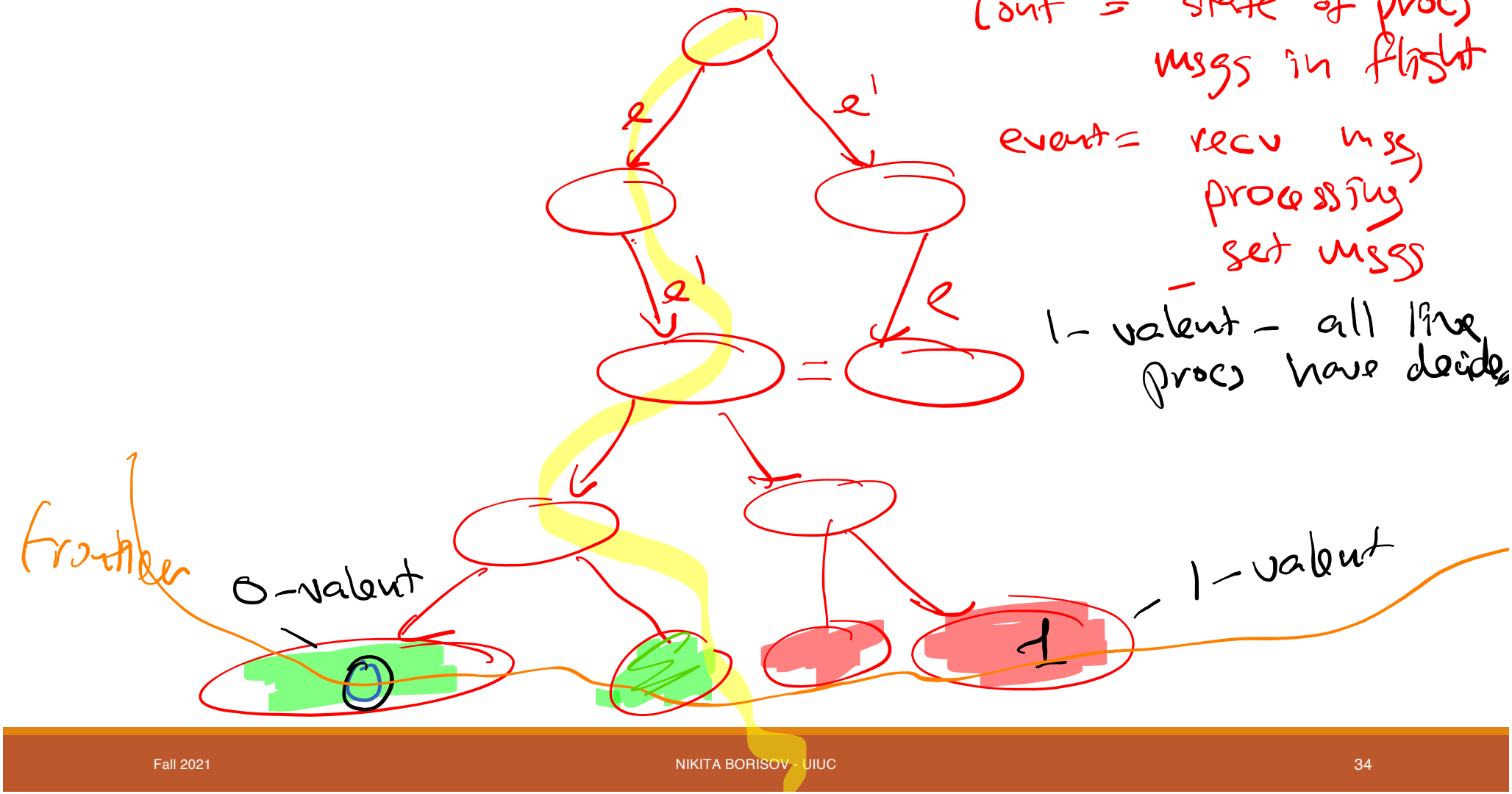

Schedules are commutative
C
Schedule s1
Schedule s2
C'
s1 and s2
•can each be applied to C
•involve
disjoint sets of receiving processes
s2
s1
C''

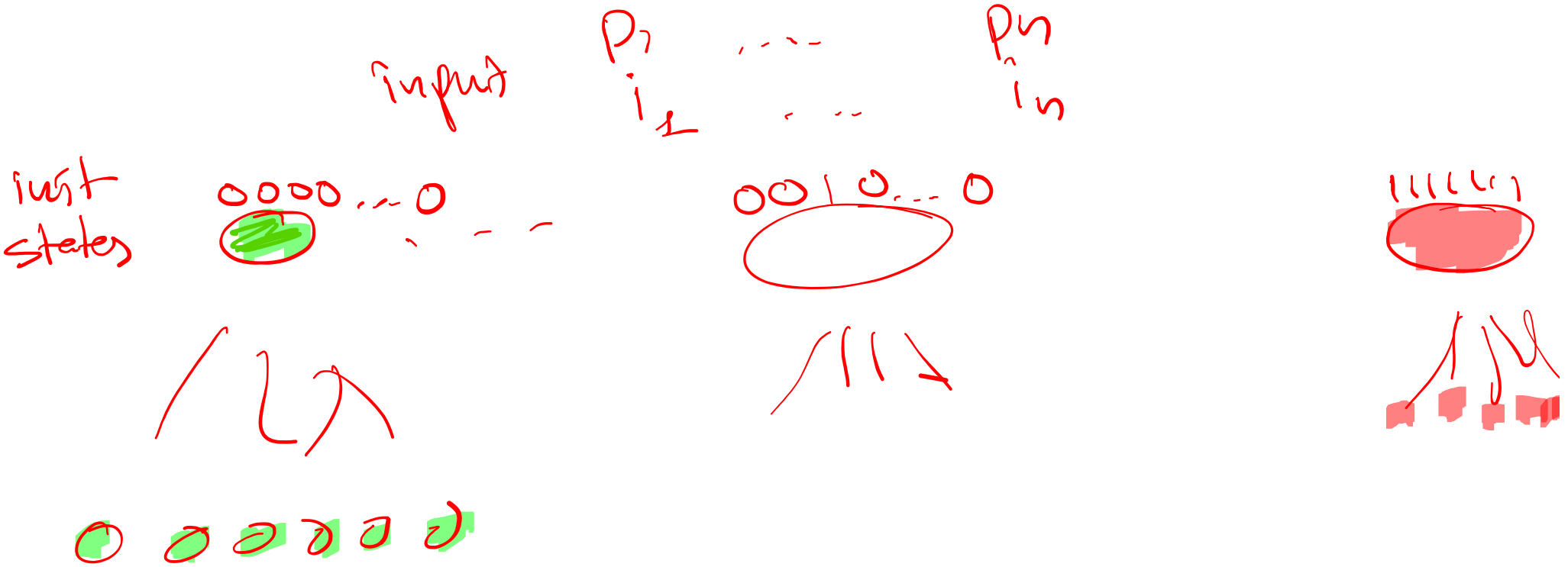
State Valencies

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 said to be 0-valent or 1-valent, as is the case

Bivalent means outcome is unpredictable


Conf = state of procs
msgs in flight
event = recv msg
processing
set msgs
1-valent - all live procs have decide
e
e'
e'
e
=
frontier
0-valent
1-valent
0
1


input
inst
states

What we'll Show

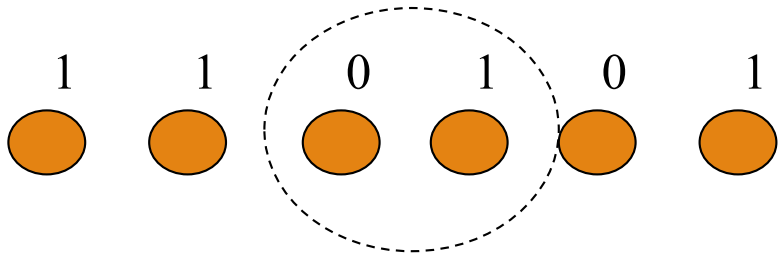
There exists an initial configuration that is bivalent

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

Lemma 2

Some initial configuration is bivalent

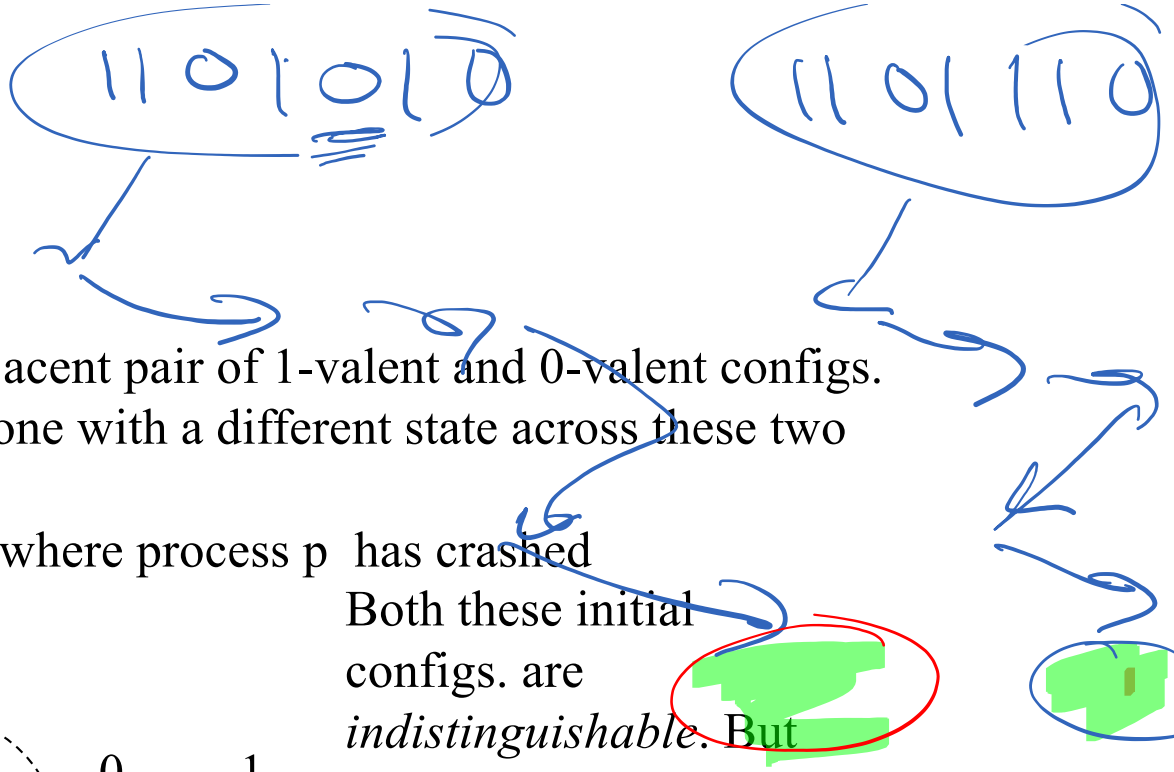
- Suppose all initial configurations were either 0-valent or 1-valent.
- Place all configurations side-by-side, where adjacent configurations differ in initial x_p value for *exactly one* process.
- Creates a lattice of states



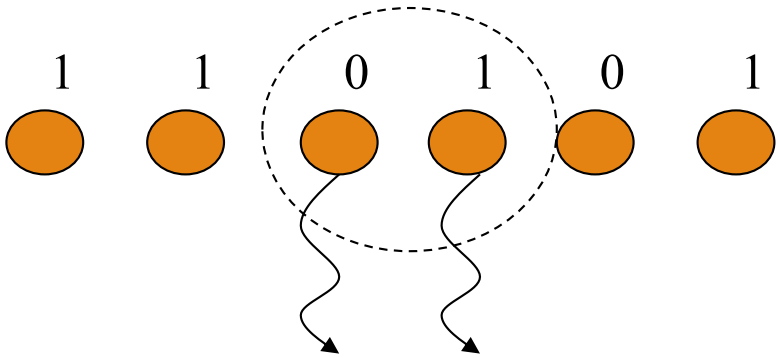
- There *has* to be some adjacent pair of 1-valent and 0-valent configs.

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 be the one with a different state across these two configs.
- Now consider the world where process p has crashed



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

So, both these initial configs. are bivalent

What we'll Show

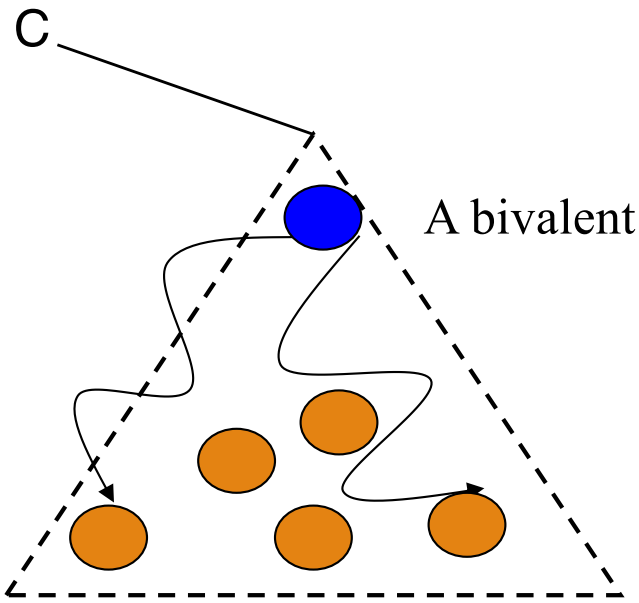
There exists an initial configuration that is bivalent

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

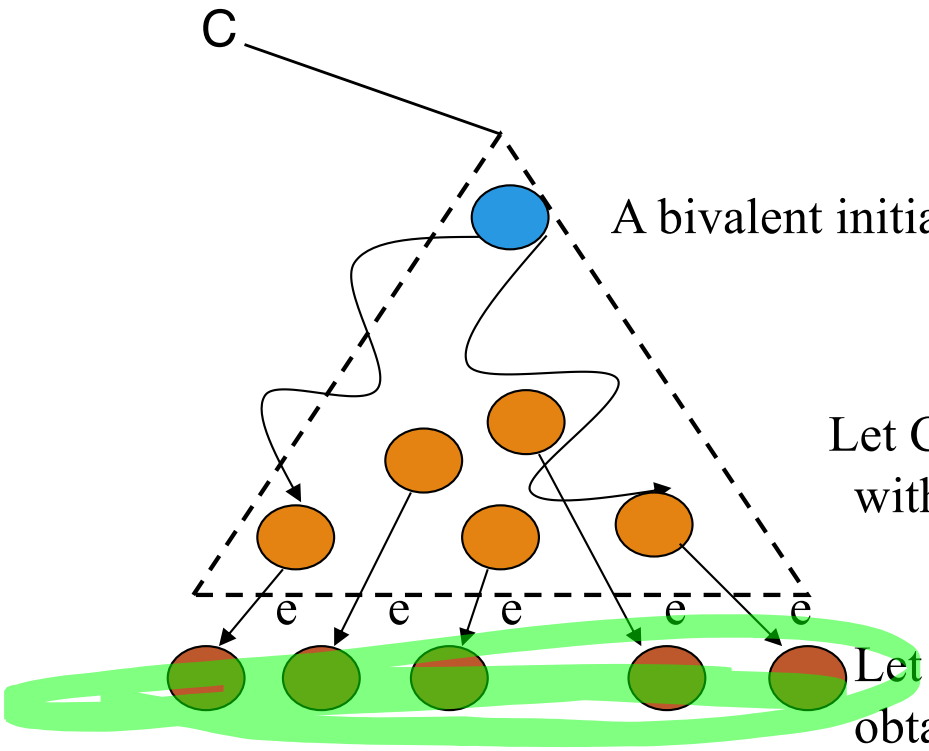


A bivalent initial config.

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

Let C be the set of configs. reachable <u>without</u> applying e

Lemma 3



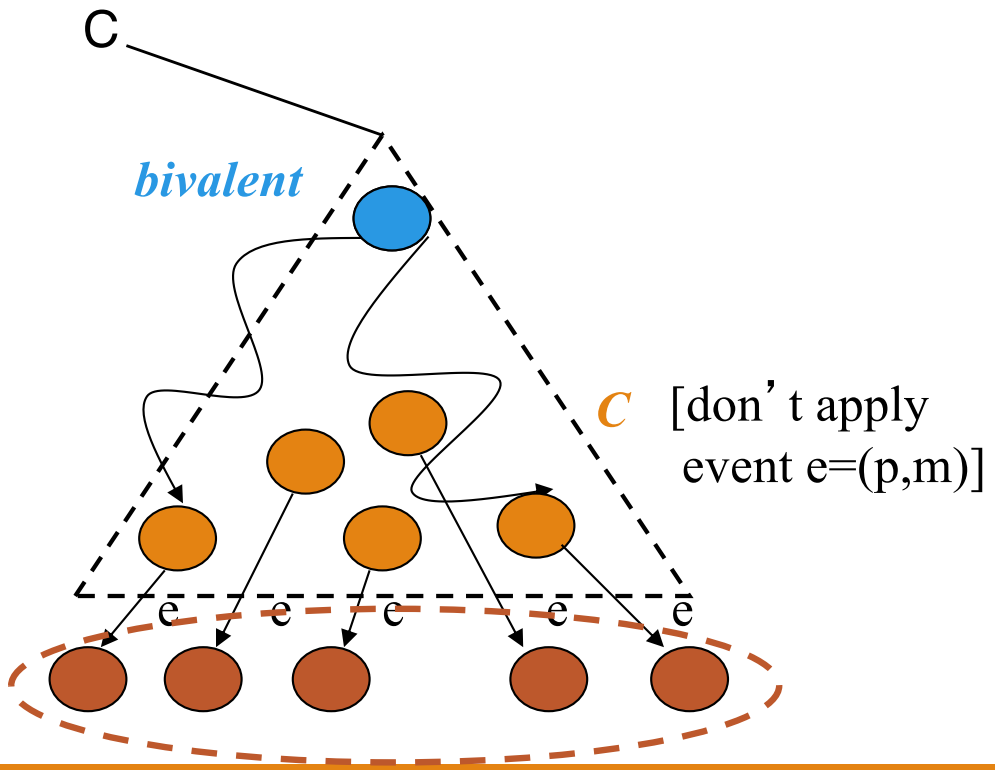
A bivalent initial config.

let e=(p,m) be an applicable event 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 single event e to any config. in ***C***

Lemma 3



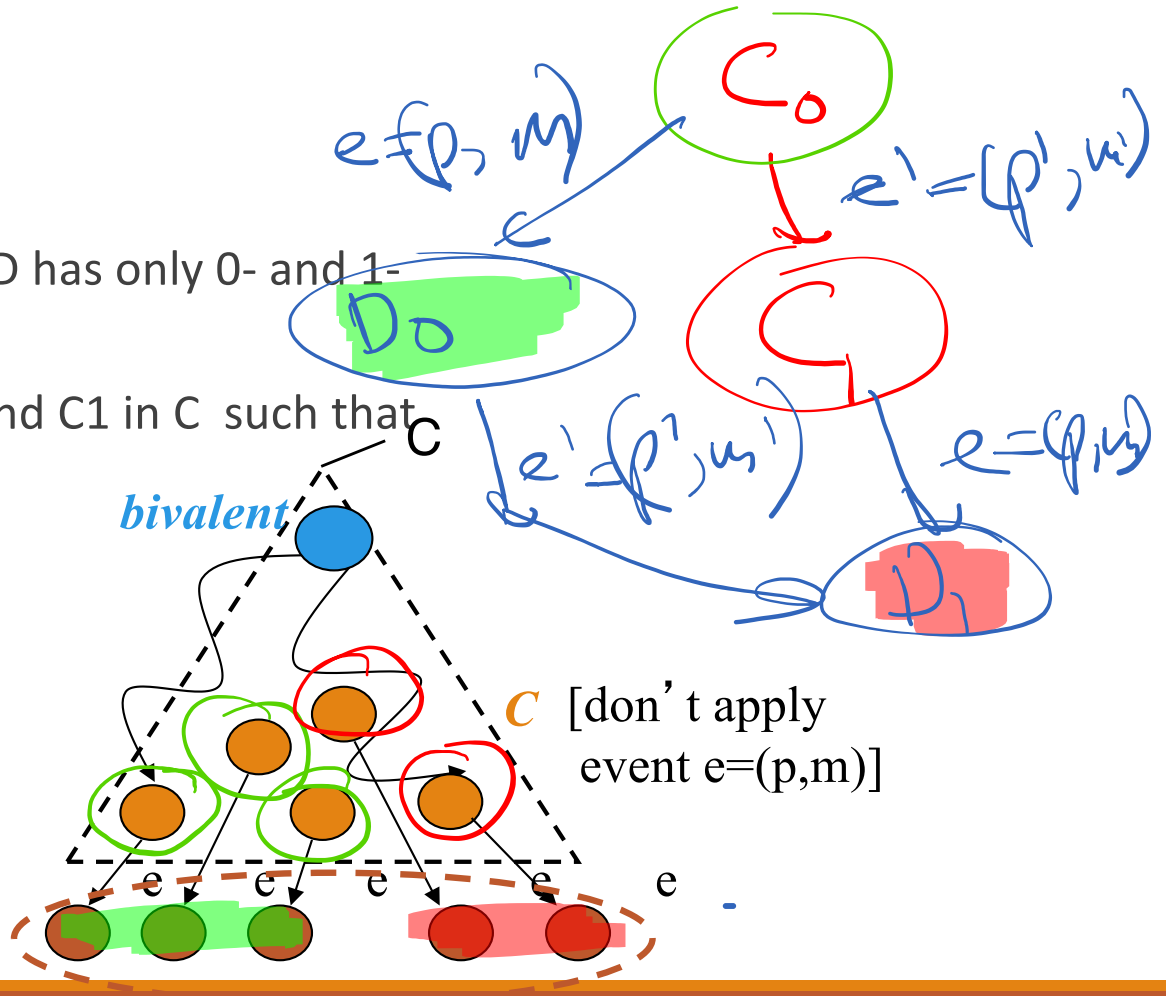
Claim. Set D contains a bivalent config.

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

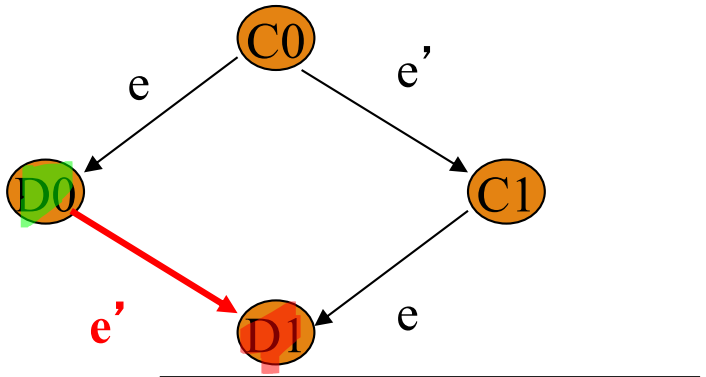
- D0 is 0-valent, D1 is 1-valent
- D0=C0 foll. by e=(p,m)
- D1=C1 foll. by e=(p,m)
- And C1 = C0 followed by some event e'=(p',m')

(why?)

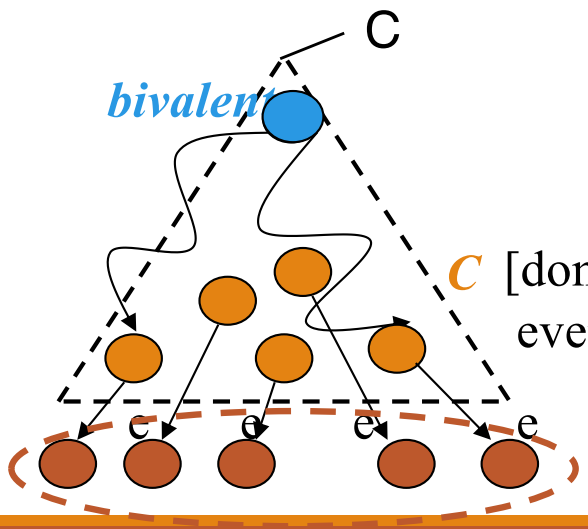


Proof. (contd.)

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



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

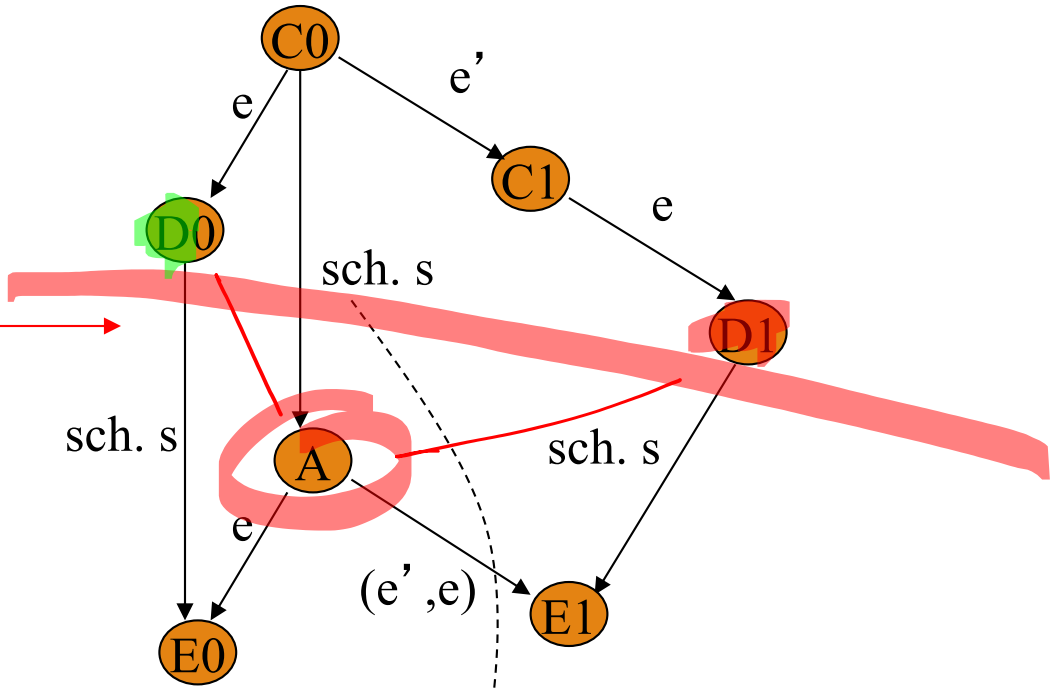


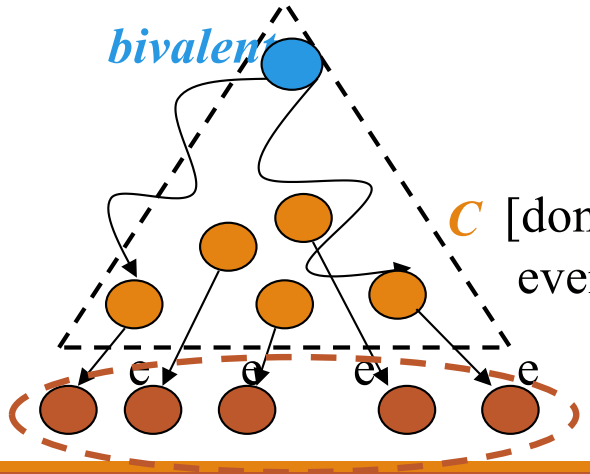
C [don't apply event e=(p,m)]

e

Proof. (contd.)

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





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

e

sch. s
- finite
- deciding run from C0 (i.e., A is not bivalent)
- 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 (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)
  - Completeness + Accuracy impossible to guarantee

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 only one process failure and arbitrarily slow processes, there are always sequences of events for the system to decide any which way. Regardless of which consensus algorithm is running underneath.
- FLP impossibility proof