Distributed Systems

CS425/ECE428

03/06/2020

Today's agenda

- Consensus
  - Consensus in synchronous systems
    - *Chapter 15.4*
  - **Impossibility of consensus in asynchronous systems**
    - *Impossibility of Distributed Consensus with One Faulty Process, Fischer-Lynch-Paterson (FLP), 1985*
  - A good enough consensus algorithm for asynchronous systems:
    - *Paxos made simple, Leslie Lamport, 2001*
  - Other forms of consensus
    - Blockchains
    - Raft (log-based consensus)

Recap

- Consensus is a fundamental problem in distributed systems.
    - Each process proposes a value.
    - All processes must agree on one of the proposed values.

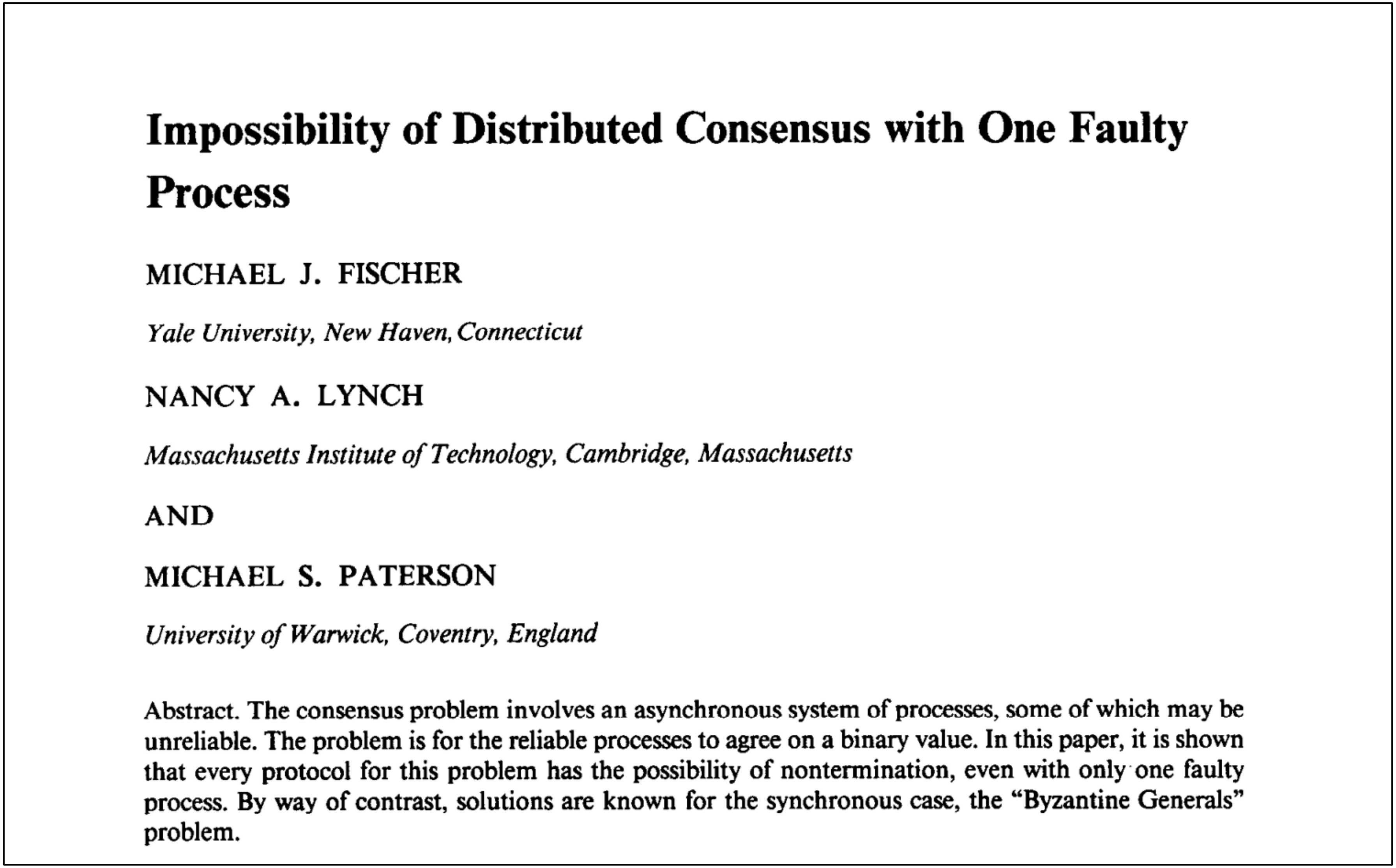
- Possible to solve consensus in synchronous systems.
    - Algorithm based on time-synchronized rounds.
    - Need at least (f+1) rounds to handle up to f failures.

- Impossible to solve consensus in asynchronous systems.
    - Paxos algorithm:
        - Guarantees safety but not liveness.
        - Hopes to terminate if under good enough conditions.
    - **Why? FLP result.**

Consensus in asynchronous systems

- Cannot use timeout-based “rounds”.
  - Do not have clocks with bounded synchronization.
- Failure detection cannot be both complete and accurate.
  - Cannot differentiate between an extremely slow process and a failed process.
- Consensus is impossible in an asynchronous system.
- Proved in the now-famous FLP result.
  - Stopped many distributed system designers dead in their tracks.
  - A lot of claims of “reliability” vanished overnight.

FLP result



Impossibility of Distributed Consensus with One Faulty Process

MICHAEL J. FISCHER

Yale University, New Haven, Connecticut

NANCY A. LYNCH

Massachusetts Institute of Technology, Cambridge, Massachusetts

AND

MICHAEL S. PATERSON

University of Warwick, Coventry, England

Abstract. The consensus problem involves an asynchronous system of processes, some of which may be unreliable. The problem is for the reliable processes to agree on a binary value. In this paper, it is shown that every protocol for this problem has the possibility of nontermination, even with only one faulty process. By way of contrast, solutions are known for the synchronous case, the "Byzantine Generals" problem.

Weaker Consensus Problem

- FLP result applicable even for a weak form of consensus problem.
  - Every process p has an input (proposed) value x_p in {0,1}.
  - Every process maintains an output value y_p initialized to *b* in the undecided state.
  - Upon entering its *decided* state, a non-faulty process sets y_p to a value in {0,1}.
    - y_p is not changed once it is set in the *decided* state.

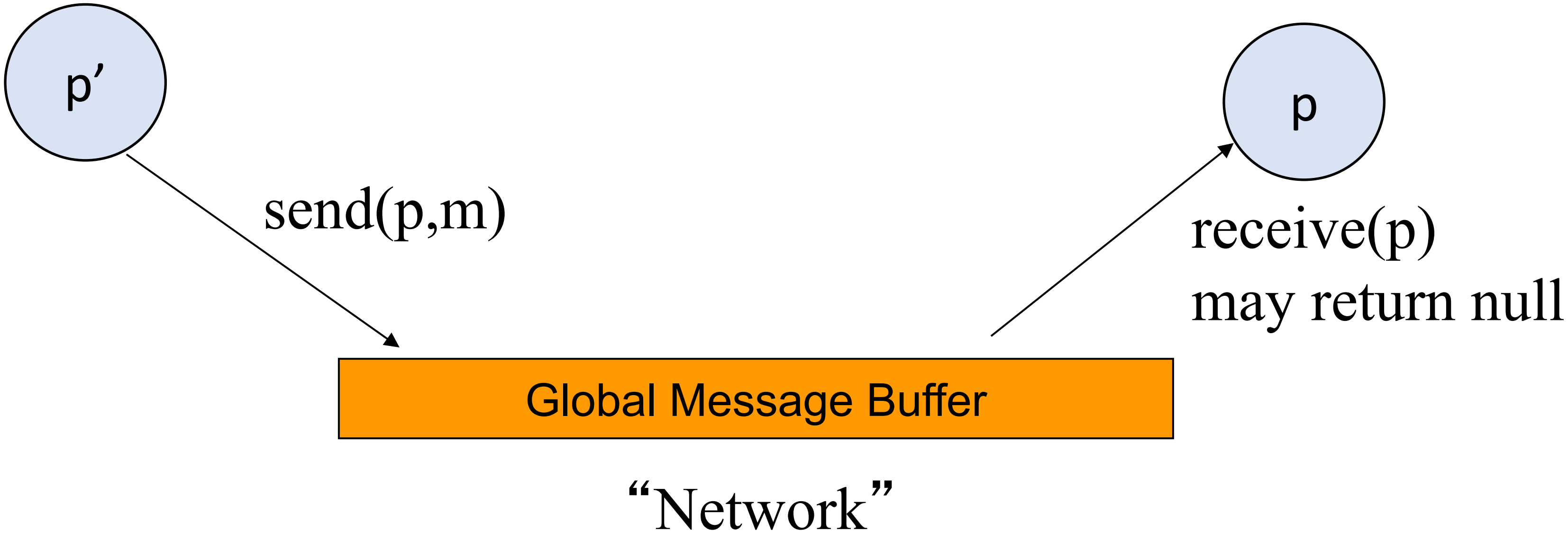
Weaker Consensus Problem

- FLP result applicable even for a weak form of consensus problem.
  - Requirements:
    - All non-faulty processes in decided state must have chosen the same value. (safety)
    - *Some* process eventually makes a decision. (liveness)
  - Trivial solution of always choosing 0 is discarded.
    - Must pick a proposed value. (validity)
    - If all processes propose '1', then chosen value must be '1'. (integrity).
    - Both 0 and 1 are possible decision values.

Assumptions

- Impossibility result holds when there is at least one process that fails by crashing (stops entirely) during the run of the consensus algorithm.
    - Let’s assume that only one process crashes (could be any one).
- Consensus protocol is deterministic.
- Message system is reliable.
    - A message will eventually get delivered.
    - Message may be arbitrarily delayed.

Message system (network) model



p’

p

send(p,m)

receive(p)
may return null

Global Message Buffer

“Network”

- Abstractly, a process p “calls” receive(p) to receive a message from the network.
- The network may return “null” a finite number of times.
- After infinite attempts of receive(p), p will receive all messages meant for it.

Notations

- **Configuration**: internal state of each process and the state of message buffer.
  - Similar notion to the *global state* of the system.
  - Initial configuration: initial state of a process and empty message buffer.

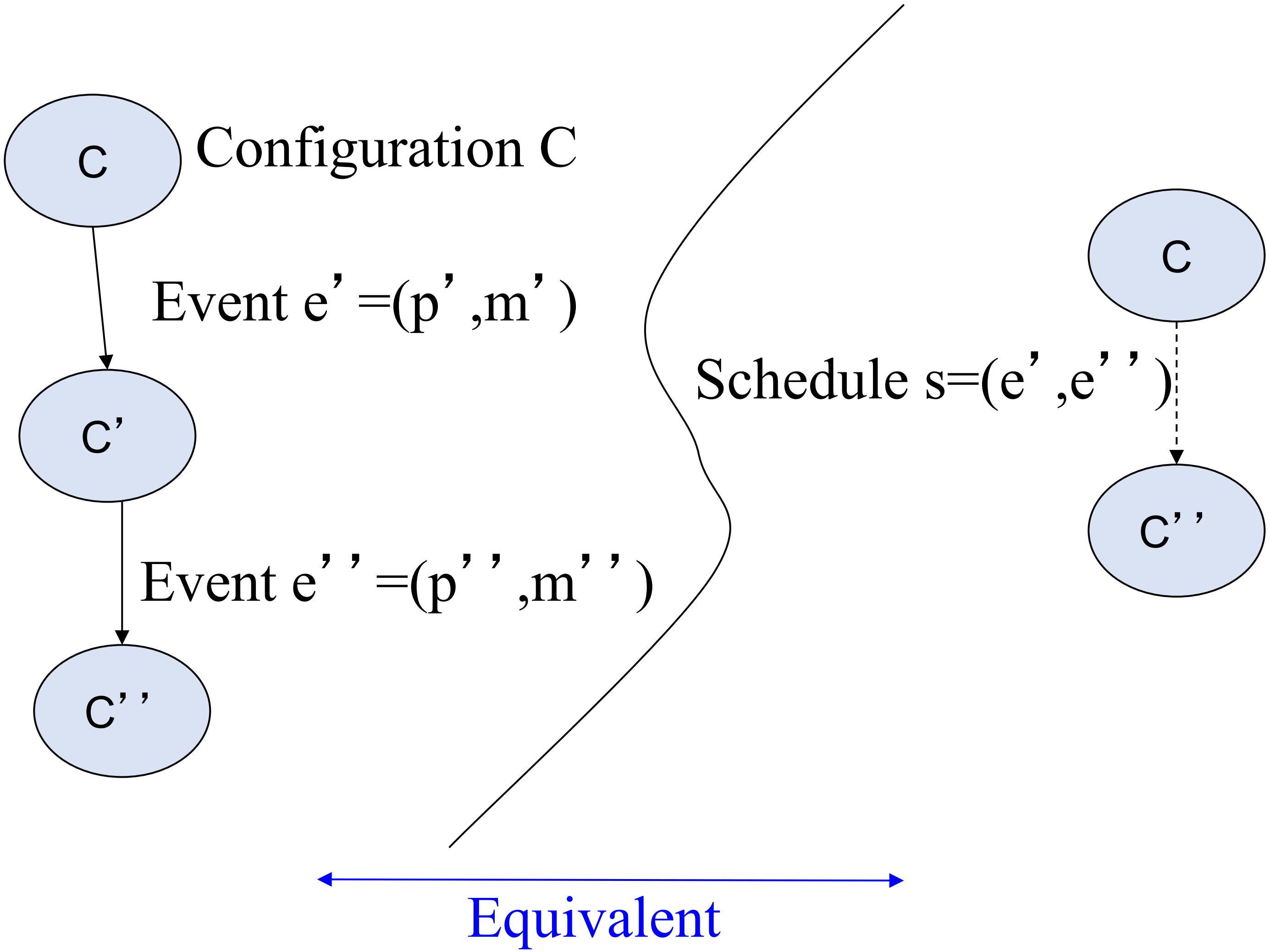
- **Event** described as e = (p, m) fully defines a *step* taken by a process in config. C
    - e = (p, m): process *p* receives message *m.* (*m* is allowed to be null).
    - Internal processing of *m* at p changes config. from C to C'.
    - p may then send a finite set of messages to other processes
- A *step* taken by process p changes configuration from one to another.

- **e(C):** the resulting configuration C' after event e is *applied* to configuration C.
  - (p, null) can always be applied to C. Always possible for p to take a step.

- **Schedule (s):** sequence of events applied to C.
  - Let *s* = $\{e_1,e_2,e_3,e_4\}$, then $s(C) = e_4(e_3(e_2(e_1(C))$
  - If *s* is finite, s(C) is *reachable* from C.

Notations

- **Schedule (s):** sequence of events applied to C.



C

Configuration C

Event e’ =(p’ ,m’)

C’

Event e’ ’ =(p’ ’ ,m’ ’)

C’ ’

C

Schedule s=(e’ ,e’ ’)

C’ ’

Equivalent

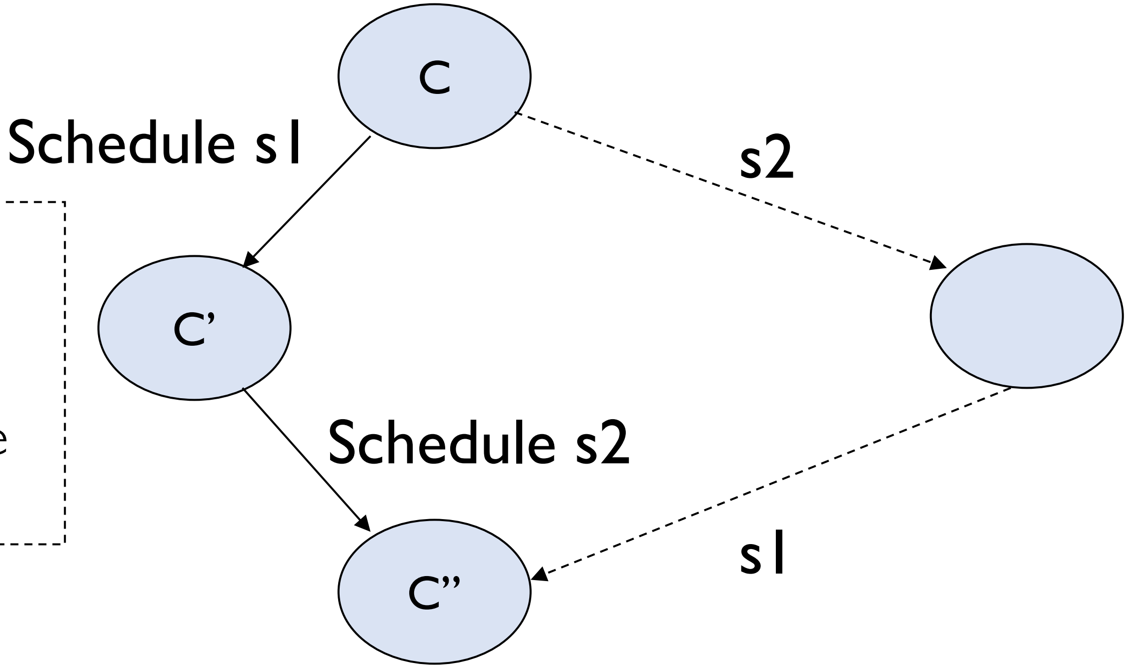
Notations

- **Schedule (s):** sequence of events applied to C.
- The associated sequence of steps in the schedule is called a *run*.
- A run is *deciding* if some process reaches a decision state in that run.

Lemma 1

Disjoint schedules are commutative.

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



Since s1 and s2 never interact, their relative ordering should not affect the final configuration.

Bivalent vs Univalent

- Let config. C have a set of decision values V reachable from it.
  - Configurations reachable from C have processes in decided state with the decided value in V.

- If |V| = 2, config. C is bivalent
- If |V| = 1, config. C is univalent
  - 0-valent or 1-valent, as is the case

- Bivalent means outcome is unpredictable.

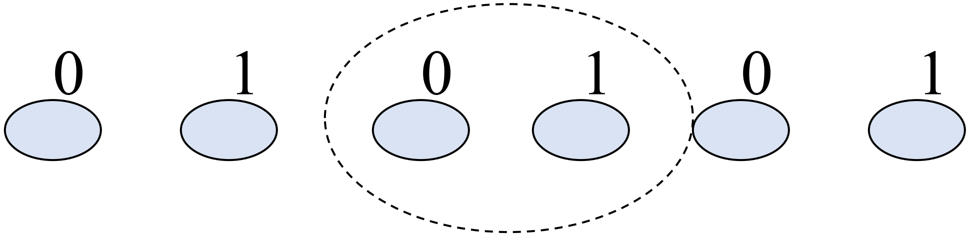
What we will show

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 2

Some initial configuration is bivalent

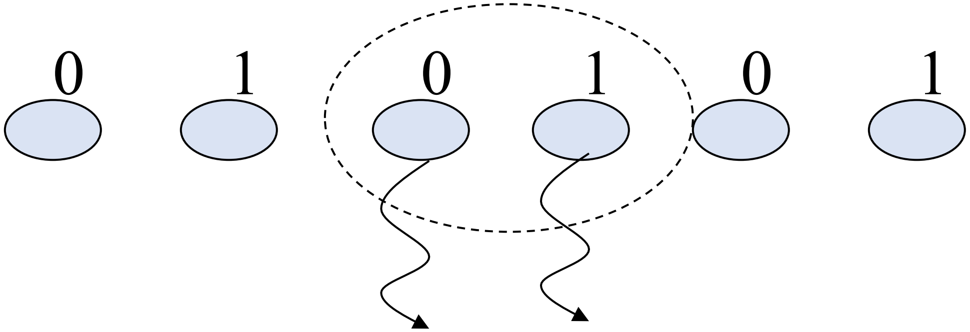
- Suppose all initial configurations were either 0-valent or 1-valent.
- If there are N processes, there are 2^N possible initial configurations
- Place all configurations side-by-side (in a lattice), where adjacent configurations differ in initial x_p value for exactly one process.
- Both 0-valent and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

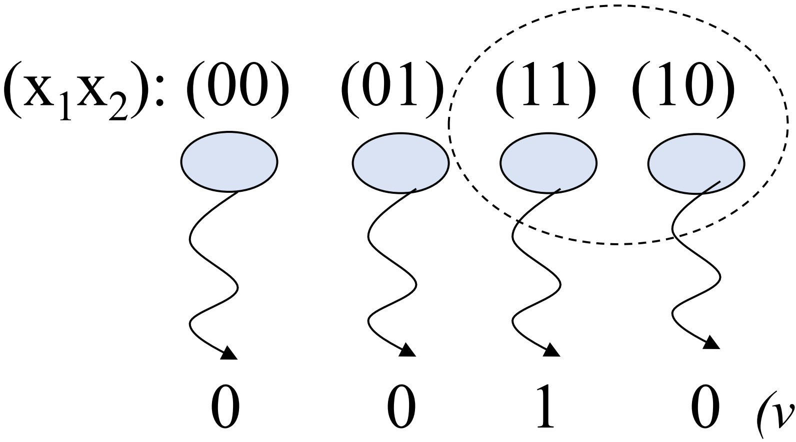


- Under such a failure, both initial configs. will lead to the same config. for the same sequence of events.
- Therefore, at least one of these initial configs. is bivalent when there is such a failure.

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)



(valency without failures)

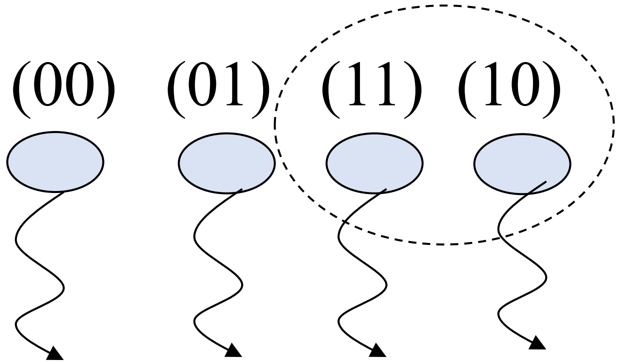
Example: system of two process.
Algorithm sets $y_p = \min(x_1, x_2)$.
What if p_2 never sends a message?

- Under such a failure, both initial configs. will lead to the same config. for the same sequence of events.
- Therefore, at least one of these initial configs. is bivalent when there is such a failure.

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)



(x_1x_2): (00) (01) (11) (10)

0 0 1 b *(if p_2 never sends a message)*

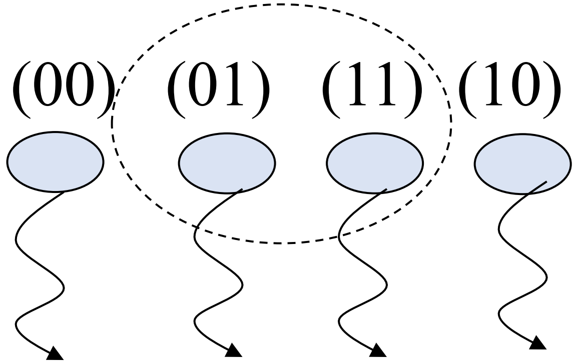
Example: system of two process.
Algorithm sets $y_p = \min(x_1, x_2)$.
What if p_2 never sends a message?

- Under such a failure, both initial configs. will lead to the same config. for the same sequence of events.
- Therefore, at least one of these initial configs. is <u>bivalent</u> when there is such a failure.

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)



(x_1x_2): (00) (01) (11) (10)

0 b 1 0 *(if p_1 never sends a message)*

Example: system of two process.

Algorithm sets $y_p = \min(x_1, x_2)$.

What if p_1 never sends a message?

- Under such a failure, both initial configs. will lead to the same config. for the same sequence of events.
- Therefore, at least one of these initial configs. is bivalent when there is such a failure.

What we will show

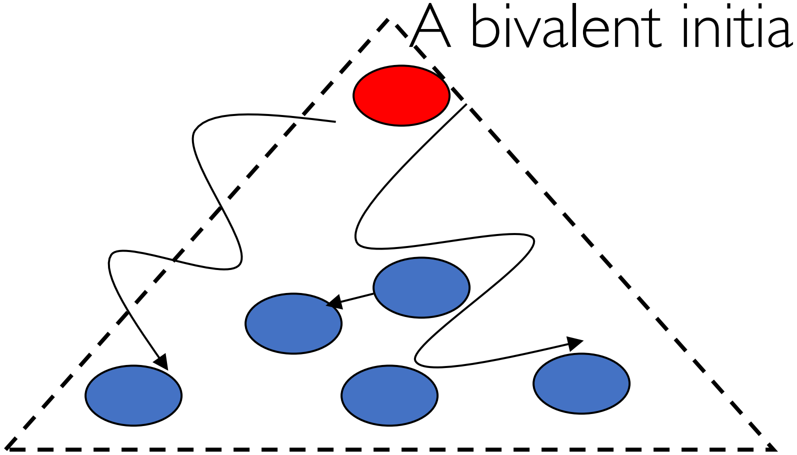
1. There exists an initial configuration that is bivalent
2. Starting from a bivalent config., there is always another bivalent config. that is reachable.

Lemma 3

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

Lemma 3

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



A bivalent initial config.

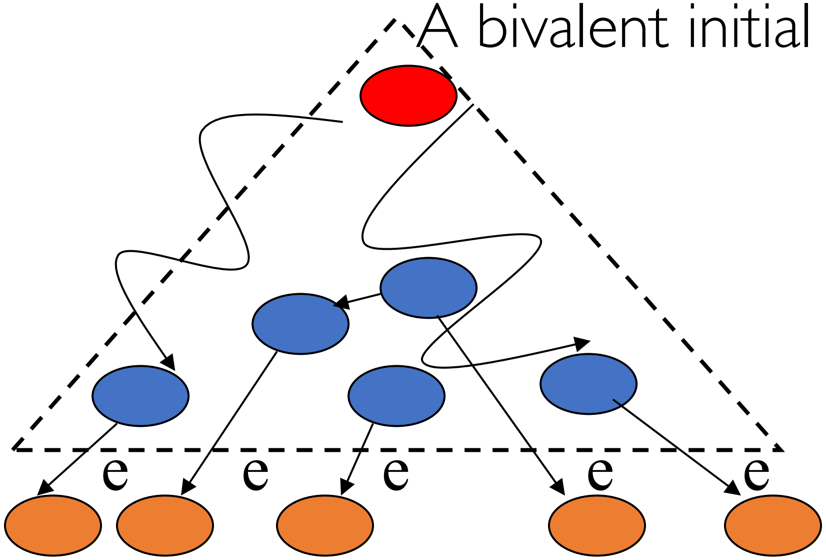
Let e=(p,m) be some event applicable to the initial config.

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

Since e is applicable to initial config., it can be arbitrarily delayed and applied to each config in **C**.

Lemma 3

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



A bivalent initial config.

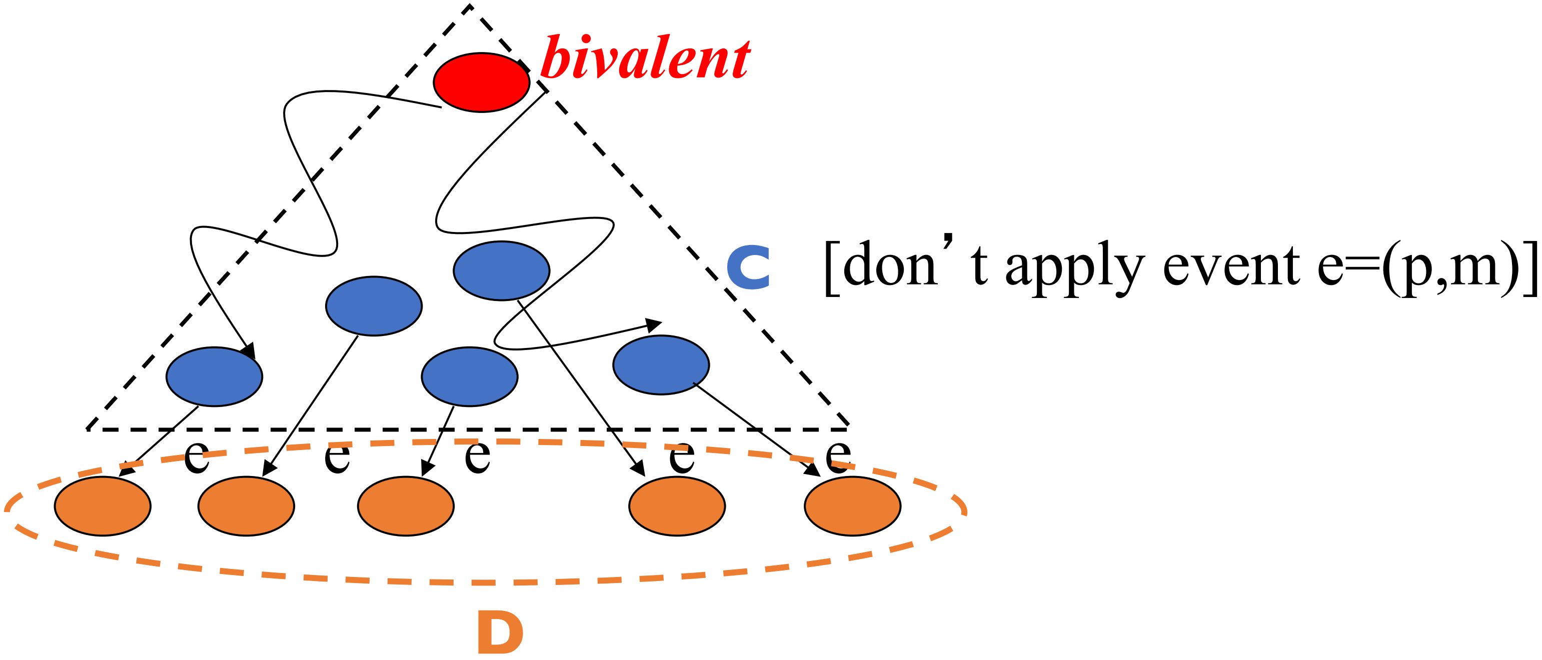
Let e=(p,m) be some event applicable to the initial config.

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

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

Lemma 3

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



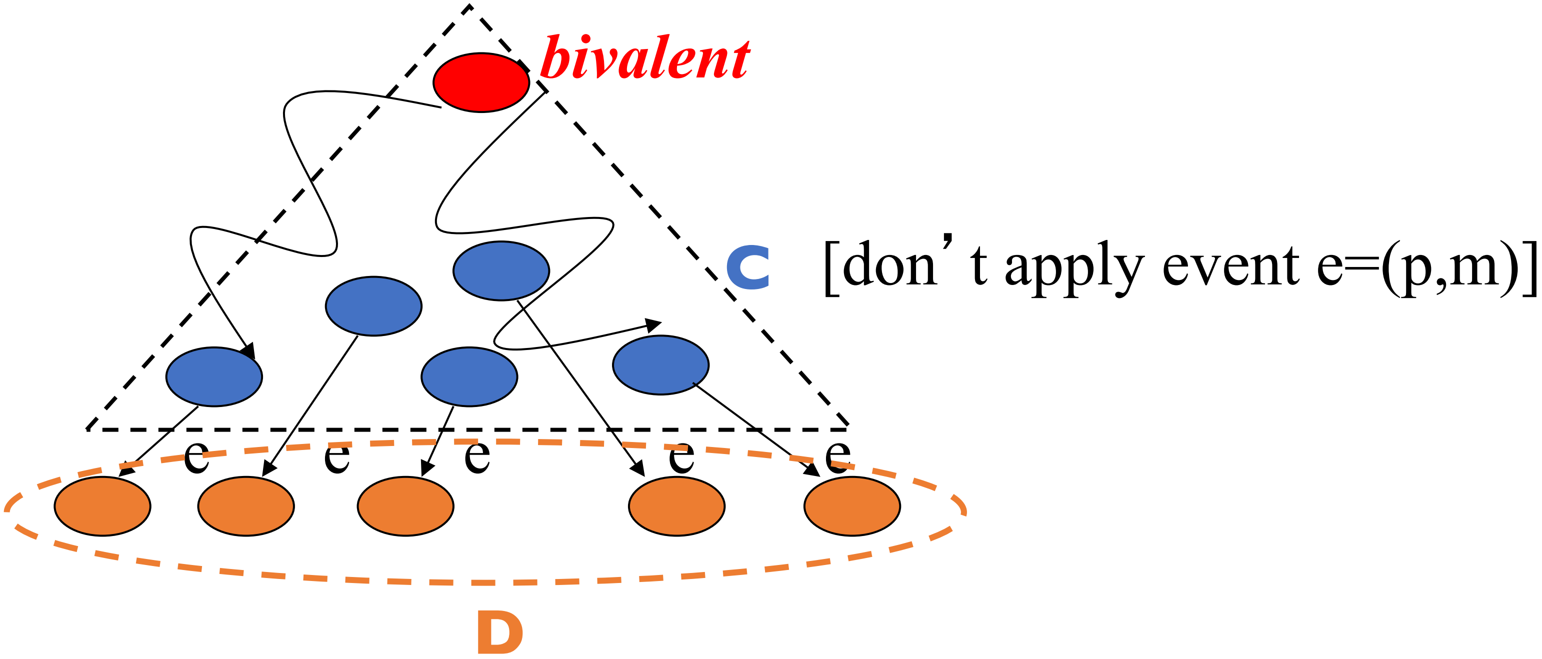
Lemma 3

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

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

We will prove this by contradiction.

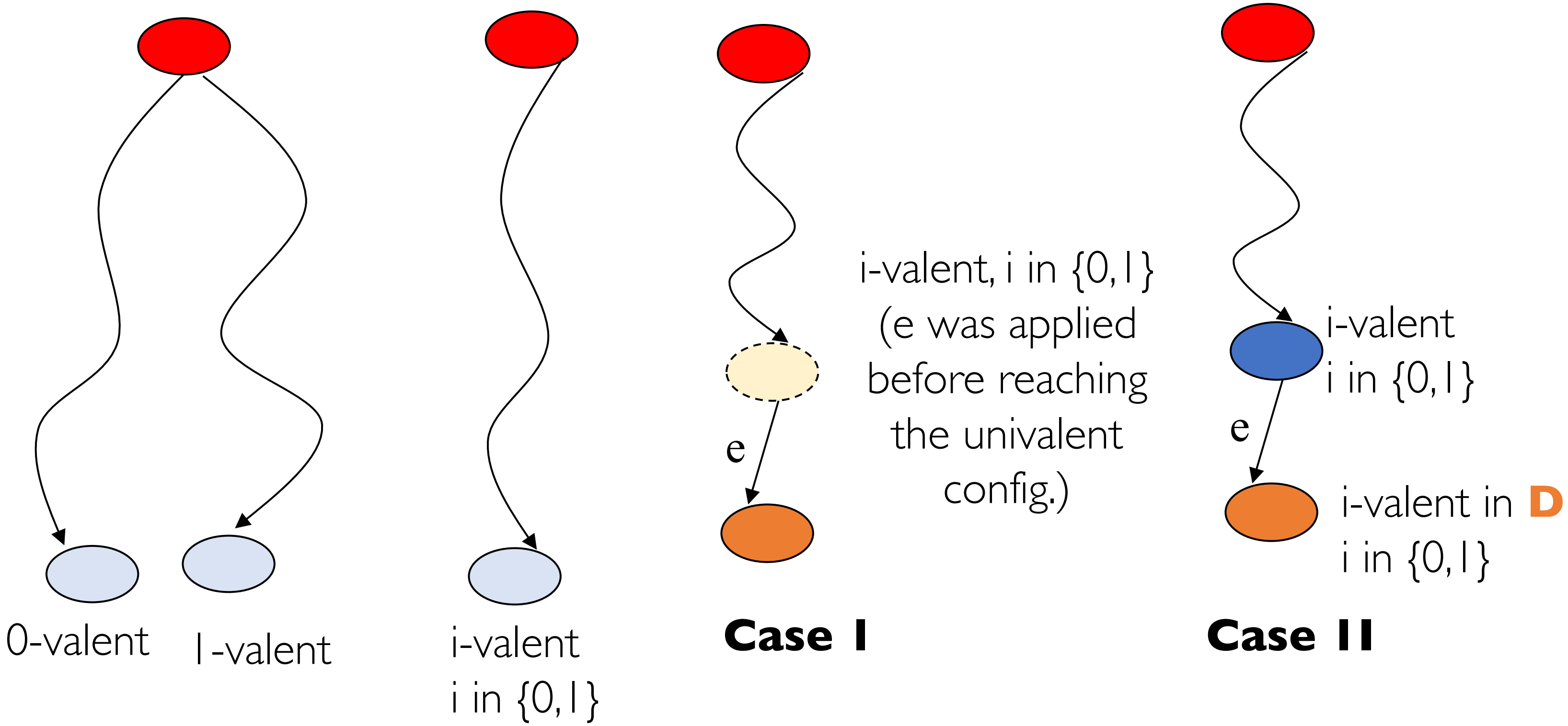
Suppose all configurations in **D** are univalent (0-valent or 1-valent).



Lemma 3

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

Suppose all configurations in **D** are univalent (0-valent or 1-valent).
D must have both a 0-valent and a 1-valent configuration.



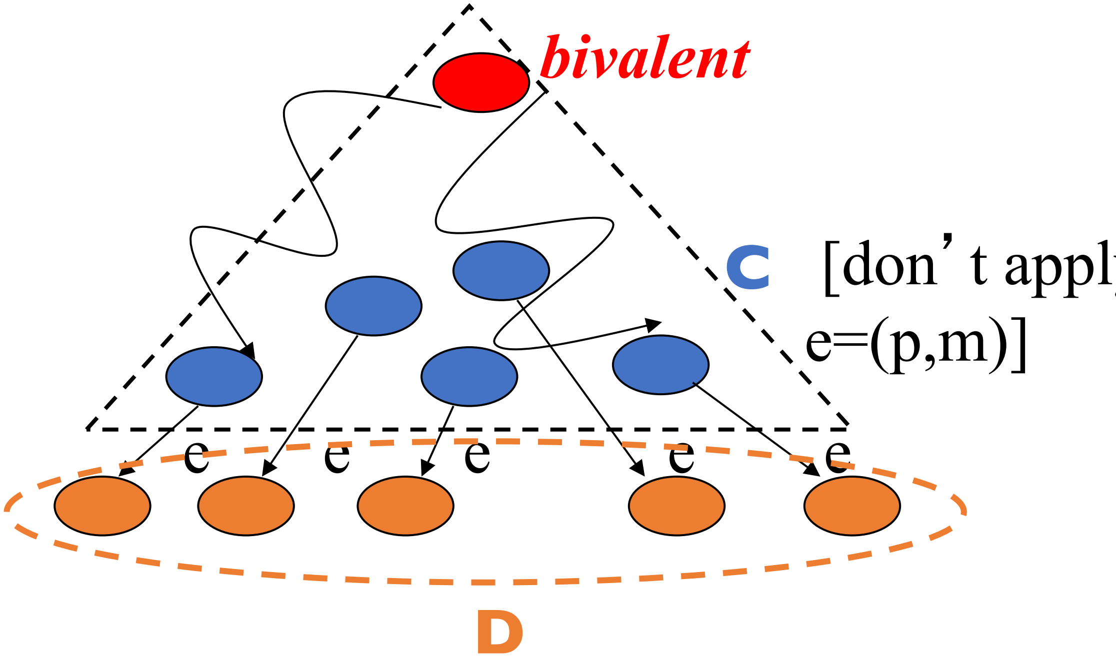
Lemma 3

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

All configs in **C** are reachable from the initial config.

We can apply e to each config in **C**.

D must have both a 0-valent and a 1-valent configuration.

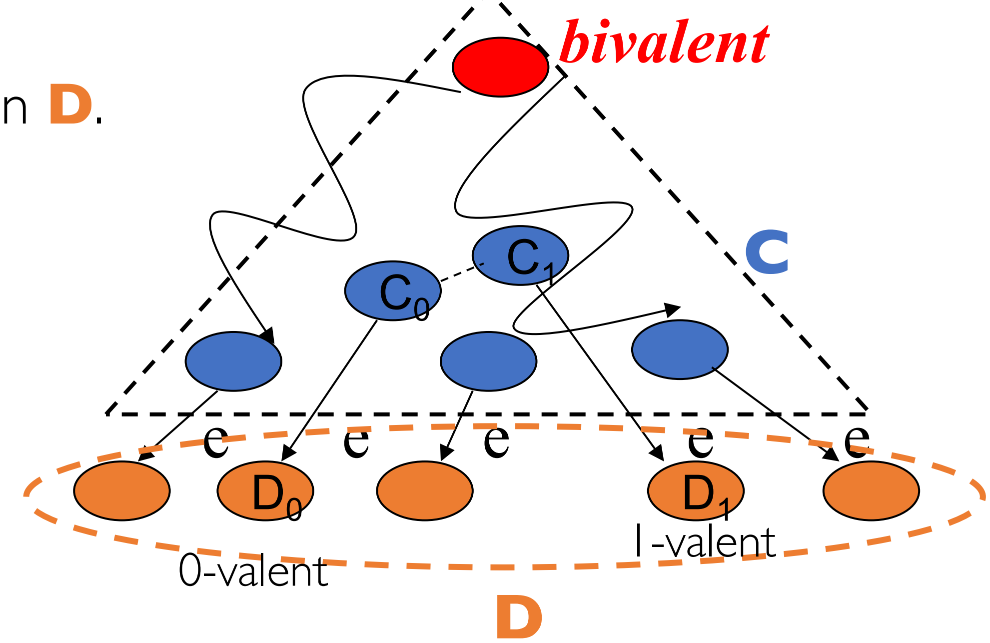


Lemma 3

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

All configs in **C** are reachable from the initial config.
We can apply e to each config in **C**.
D must have both a 0-valent and a 1-valent configuration.

There must be some *neighbouring* pair (C_0, C_1) in **C**,
such that e(C_0) = D_0 and e(C_1) = D_1
where D_0 and D_1 are
0-valent and 1-valent configs. in **D**.



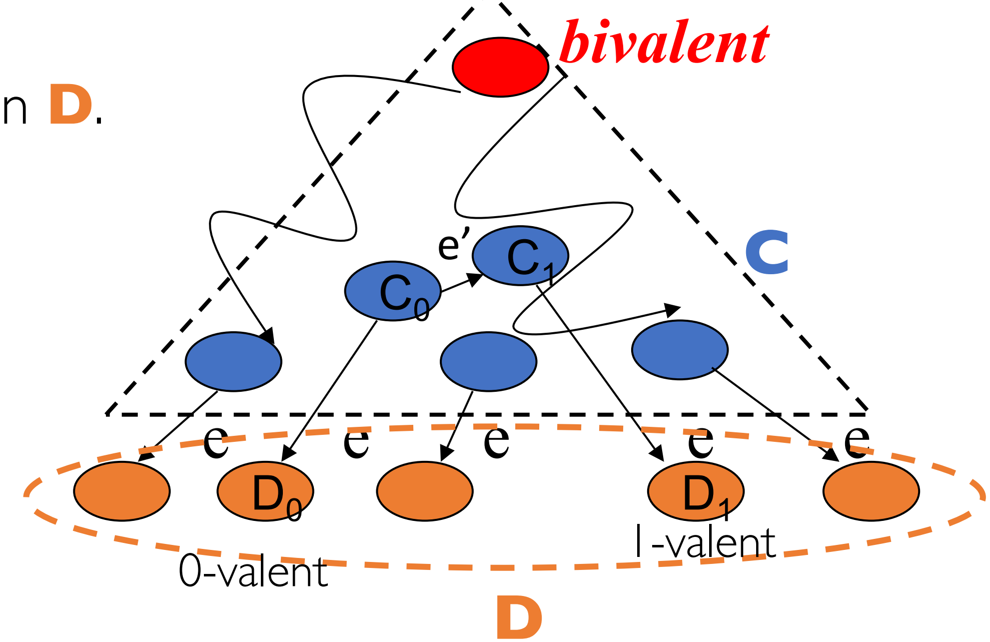
Lemma 3

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

All configs in **C** are reachable from the initial config.
We can apply e to each config in **C**.
D must have both a 0-valent and a 1-valent configuration.

There must be some *neighbouring* pair (C_0, C_1) in **C**,
such that $e(C_0) = D_0$ and $e(C_1) = D_1$
where D_0 and D_1 are
0-valent and 1-valent configs. in **D**.

Without loss of generality,
suppose $e'(C_0) = C_1$.
(could have instead assumed
$e'(C_1) = C_0$. Proof structure
will be the same.)



Lemma 3

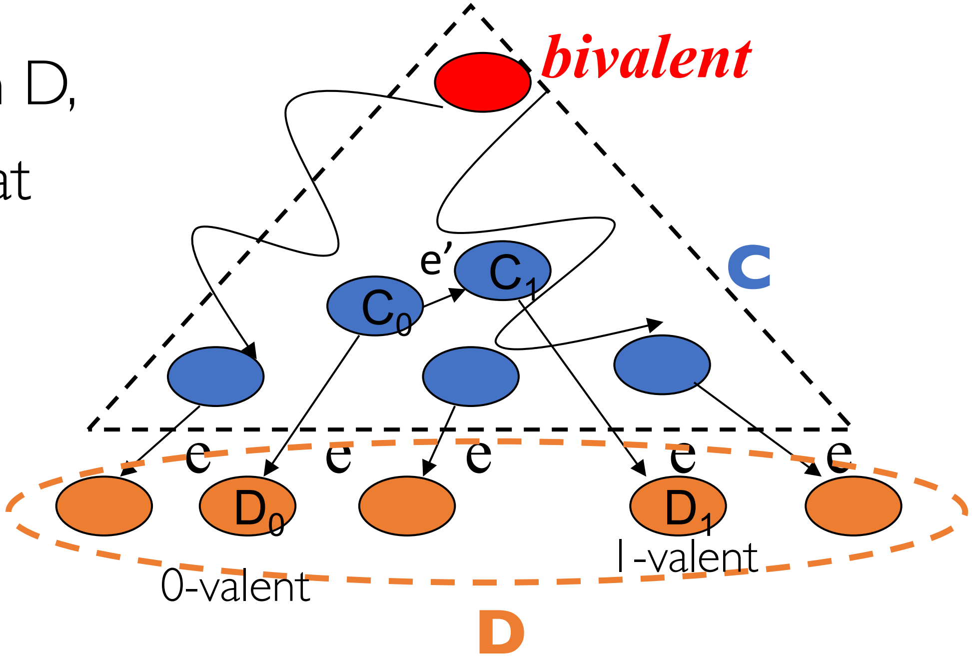
Starting from a bivalent config., there is always another bivalent config. that is reachable

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

Proof by contradiction.

- Suppose D has only 0- and 1- valent states (and no bivalent ones).
- There are states D_0 and D_1 in D,

  and C0 and C1 in C such that
  - D_0 is 0-valent
  - D_1 is 1-valent
  - $D_0=e(C_0)$, $D_1=e(C_1)$
  - $C_1 = e'(C_0)$



Lemma 3

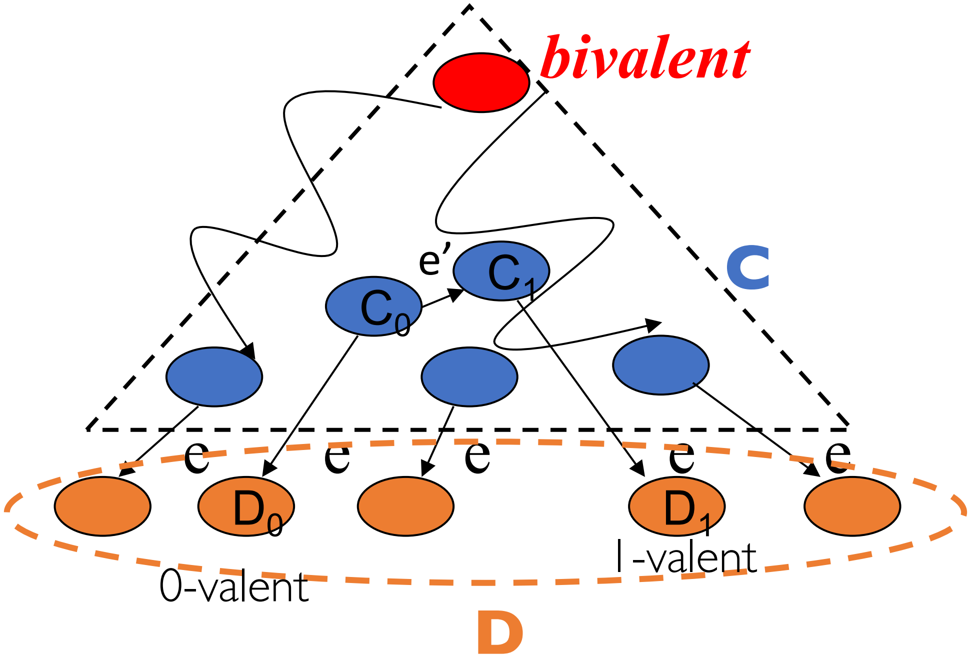
Starting from a bivalent config., there is always another bivalent config. that is reachable

Proof. (contd.)

Let e' = (p', m')

We know that e = (p, m)

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

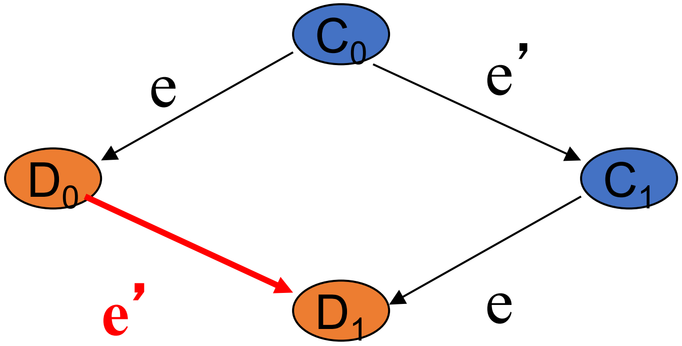


Lemma 3

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

Proof. (contd.)

- Case I: p’ is not p



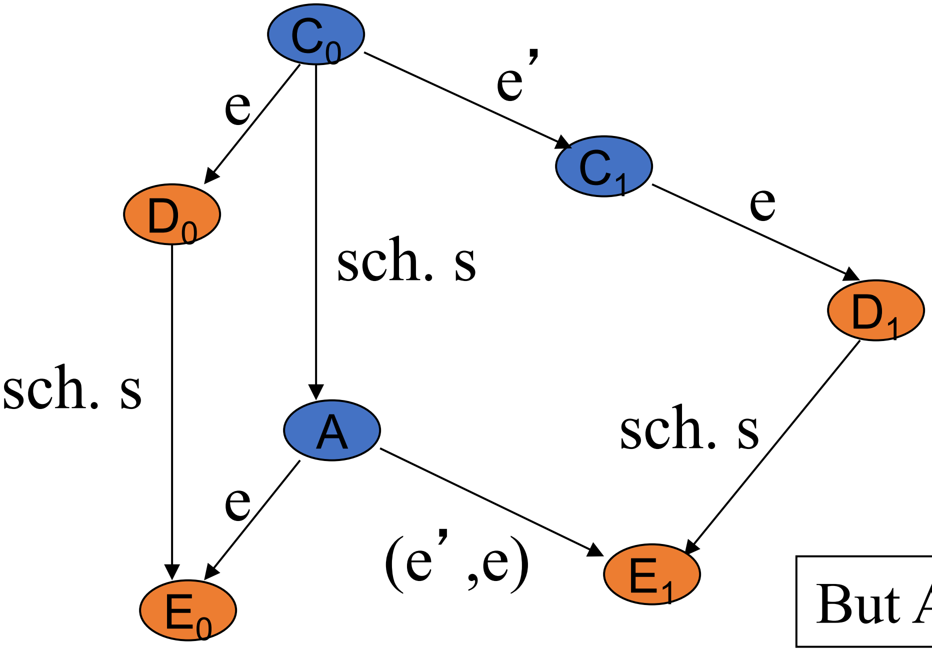
Why? (Lemma 1)
But D0 is then bivalent!

Lemma 3

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

Proof. (contd.)

- Case II: p' is p



sch. s

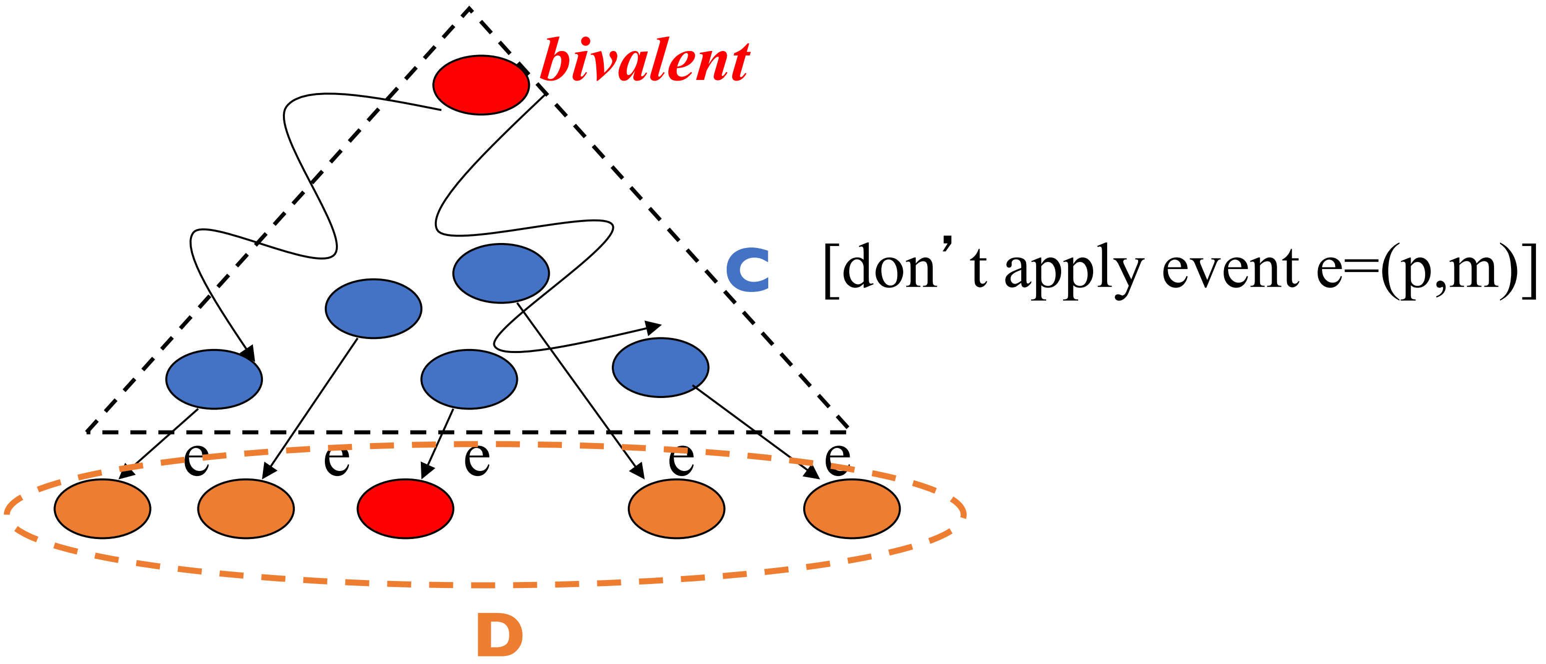
- finite
- *deciding run* from C0
    - must be univalent.
- p takes no steps

But A is then bivalent! Contradiction!

Lemma 3

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

Claim. Set **D** contains a bivalent config.
Proved by contradiction.



Putting it 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).

Putting it together

- Reaching a decision requires transitioning from a bivalent config to a univalent config.
  - A single step leads the system from a bivalent config. to a univalent config.
  - It is always possible to avoid such steps, keeping the system configs. bivalent throughout.

1. Start from a bivalent initial config. C_{init} (this exists as per Lemma 2).
2. Consider an event e = (p,m) that can be applied to C_{init}. There is a bivalent config. C_{bi} reachable from C_{init} where e is the last event applied (as per Lemma 3). Apply the corresponding sequence of events to reach C_{bi} from C_{init}.
3. Repeat from Step 1, setting $C_{init} = C_{bi}$.

Putting it 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 is a fundamental problem in distributed systems.
  - Each process proposes a value.
  - All processes must agree on one of the proposed values.

- Possible to solve consensus in synchronous systems.
  - Algorithm based on time-synchronized rounds.
  - Need at least (f+1) rounds to handle up to f failures.

- Impossible to solve consensus is asynchronous systems.
  - **FLP result.**
  - Paxos algorithm:
    - Guarantees safety but not liveness.
    - Hopes to terminate if under good enough conditions.

Next week

- Other forms of consensus:
  - Blockchains
  - Raft algorithm