ECS 265 Paper Presentation

By: Vinoth, Eva, Sandhya, Jim, Theo

Time, Clocks, and the Ordering of Events in a Distributed System

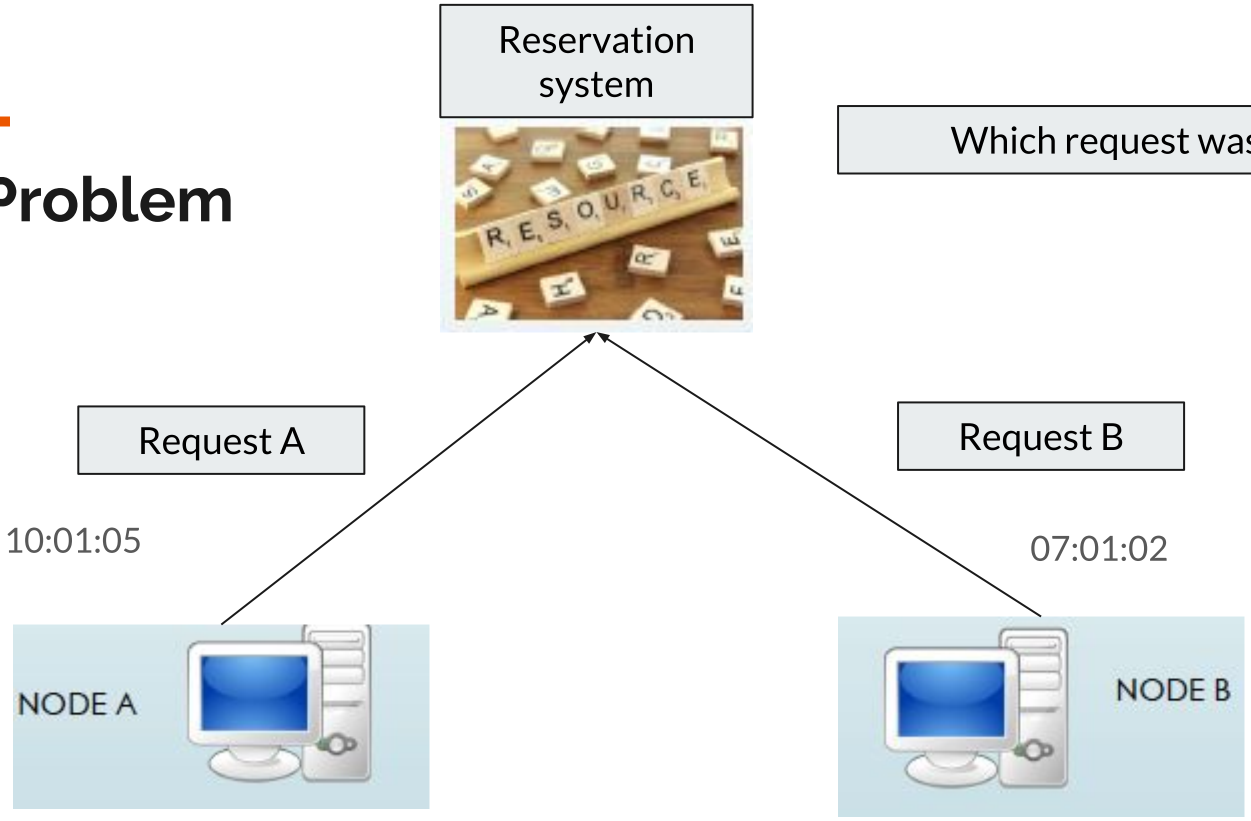
Author: Leslie Lamport

Goal of the paper:

Not to rely on Global clocks.

Order the events based on Logical clock

The Problem



Which request was made first?

The Problem

In a distributed **system** (many computers/processes connected by messages):

- There is **no single global clock**.
- Message delivery takes time, and may be delayed or reordered.
- So, different processes can have **different views of the order of events**.

The Problem

This creates ambiguity:

1. How do we decide **which event happened first**?
2. How do we **synchronize actions** (e.g., granting access to a shared resource like a database, or booking the last seat on a flight)?
3. How do we avoid contradictions like two people both getting “the last ticket”?

👉 **Main Problem:** How can we **define time and order of events** in a distributed system **without relying on perfectly synchronized physical clocks**?

Lamports proposed solution Overview:

- ➔ Partial Ordering
- ➔ Logical clocks
- ➔ Total ordering
- ➔ Mutual Exclusion Algorithm
- ➔ Anomalous behavior
- ➔ Physical clocks

Partial Ordering: ‘Happened Before’

- Goal: Define 'happened before' without relying on physical clocks.

Definition of →:

- 1. Same process ordering: a → b if a occurs before b in same process.
- 2. Message passing: send(a) → receive(b).
- 3. Transitivity: a → b and b → c implies a → c.
- Concurrency: a and b are concurrent if neither a→b nor b→a.(no order)

Concurrent events are what make the order **partial**

Visualizing the Partial Ordering

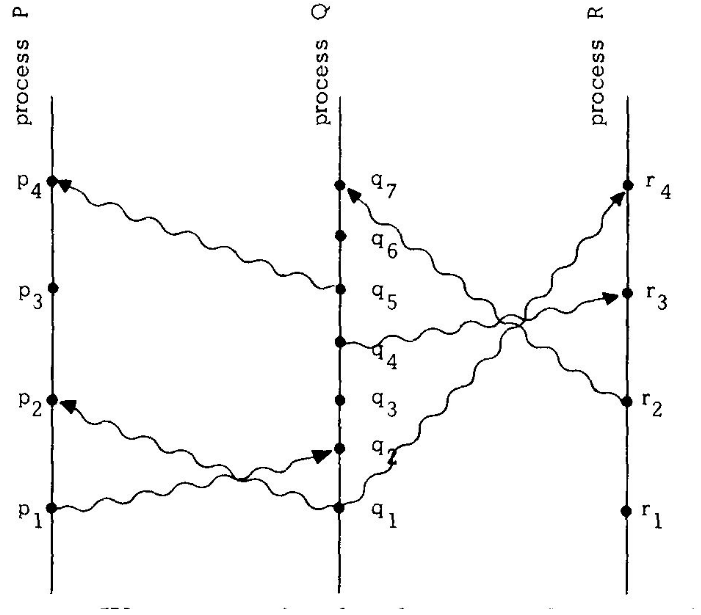
- Space-time diagram with vertical process lines and horizontal time axis.
- Dots = events; Wavy lines = messages.
- Causality: a→b means a can causally affect b.

  Example: p1 → r4.

- Concurrent events cannot causally affect each other.

  P3 and Q3 are concurrent

Fig. 1.



Logical Clocks: Assigning Time to Events

Since there's no universal clock, Lamport proposes **logical clocks**:

- Each process has a clock **Ci** : a function which assigns a number
- Logical clock Ci has no relationship with the physical clock
- Each event (eg: a) gets assigned a number (timestamp) by its processes Ci. (e.i. Ci(a))

Clock Condition : For any events a and b:
If a → b then **C(a) < C(b)**.

The Clock Condition is satisfied if the following two conditions hold.

C1. If a and b are events in process Pi, and a comes before b, then Ci(a)< Ci(b).

C2. If a is the sending of a message by process Pi and b is the receipt of that message by process Qj, then *Ci(a)* < *Cj(b).*

Logical Clocks: Implementing Logical Clocks

Following implementation rules are proposed to satisfy the clock condition:
Rules:

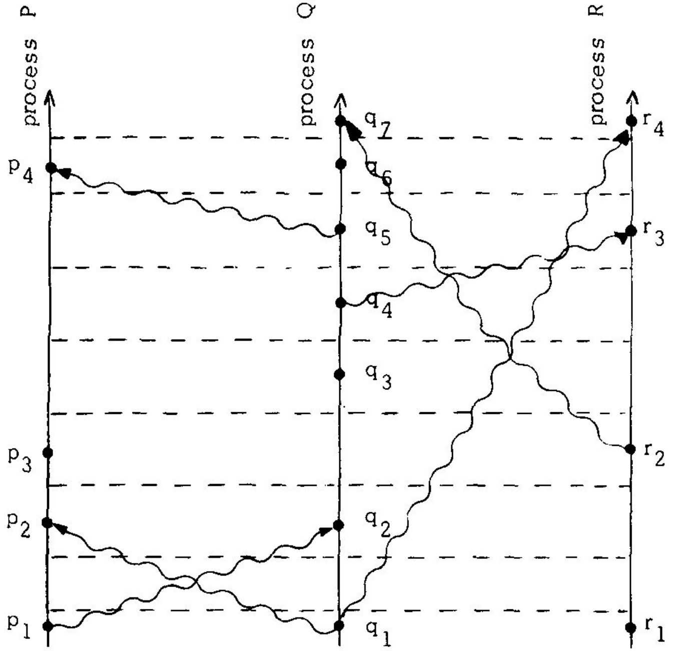
- IR1: Each process Pi increments Ci between any successive events
- IR2: if event ‘a’ is sending message ‘m’ by process pi, them message ‘m’ contains a timestamps
  - On send ‘m’: attach timestamp Tm = Ci(a)
  - On receive ‘m’: set Cj = max(Cj, Tm) + 1

Dashed horizontal lines across processes = **“tick lines” = global logical time coordinate.**

“global” = it provides one consistent frame of reference for *all* processes,

“logical” = it’s *based on event relationships (causality)*, not physical time.

Fig. 3.



Tick-Line

Fig. 1.

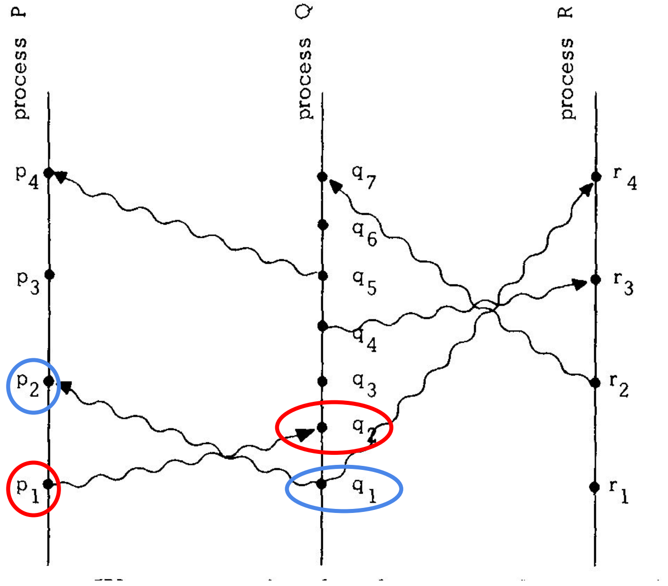
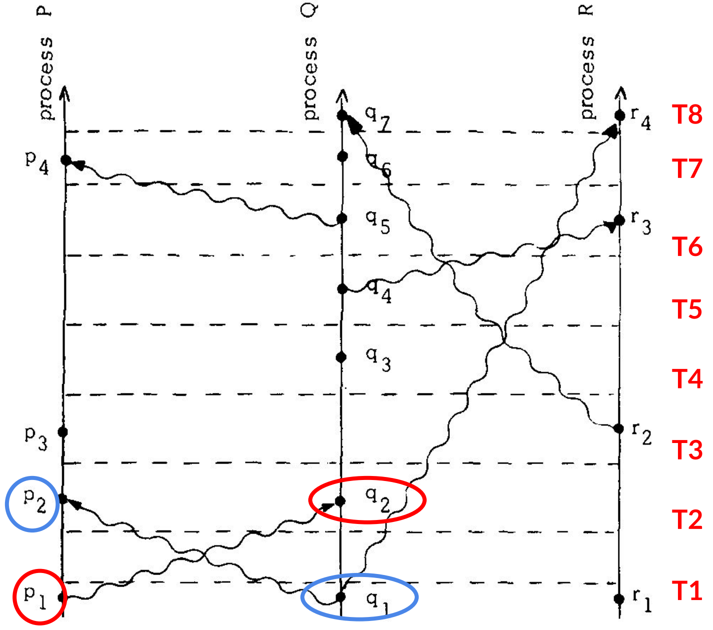


Fig. 3.



total ordering

Logical clocks give consistent partial order
For coordination, need everyone to agree on **one total sequence**
Definition:

- a ⇒ b if
  1. C(a) < C(b), or
  2. C(a) = C(b) and process ID (a) < process ID (b)

Total order = consistent global sequence for all processes
Basis for Lamport's coordination algorithms

For example:

- C(a)= C(b) = Logic time 5
- Event *a* is from P1
- Event *b* is from P3

Then the order is:

a < b (event a happens before event b) because of 1 < 3

Mutual exclusion algorithm

Goal:
Ensure only one process accesses a shared resource at a time.

Idea:
Each process maintains a **request queue** ordered by **total timestamp** (Lamport time + process ID).

Key Properties:

- **Safety**
- **Fairness**
- **Decentralization**

Key idea:
All processes agree on the same **total order**, ensuring fairness and no conflicts, without a central coordinator.

Solution to the Mutual exclusion

1. Request:

- Pi timestamps request (Tm = Ci(a))
- Sends (REQUEST, Tm, Pi) to all
- Adds to local queue

2. Receive Request:

- Pj adds request to queue
- Sends **acknowledgement** to Pi (later timestamp)

3. Release:

- Pi removes its request
- Sends (RELEASE, Pi) to all

4. Receive Release:

- Pj removes (Pi) request from queue

5. Enter Critical Section:

- Pi's request is **at the head of the queue**
- Has **acknowledgements or later messages** (messages thats already been seen/processed) from all processes

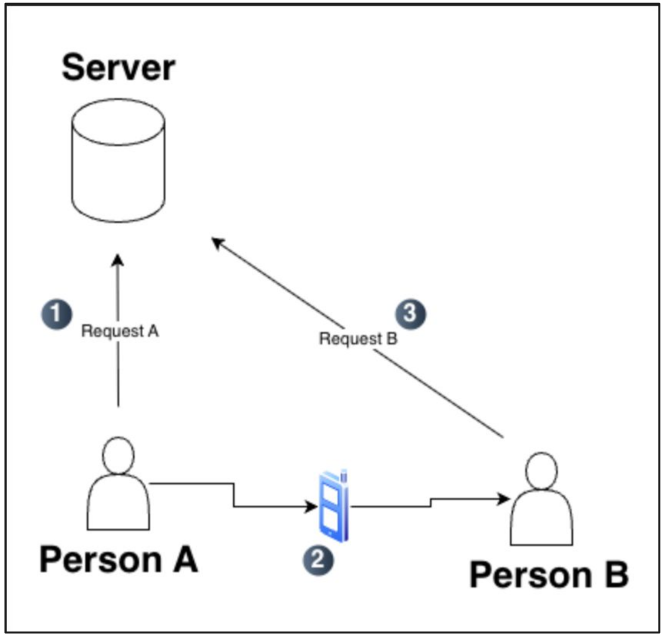
Anomalous Behavior

- System is only aware of internal messages and blind to the outside

- Mismatch between real-world causality and system-level ordering

- Occurs when total ordering of events derived from logical clocks contradicts the **causal ordering** perceived by external user (human action like phone calls).

Example



Causal Order	**ReqA -> ReqB**
Internal Order	**ReqB -> ReqA**

Example

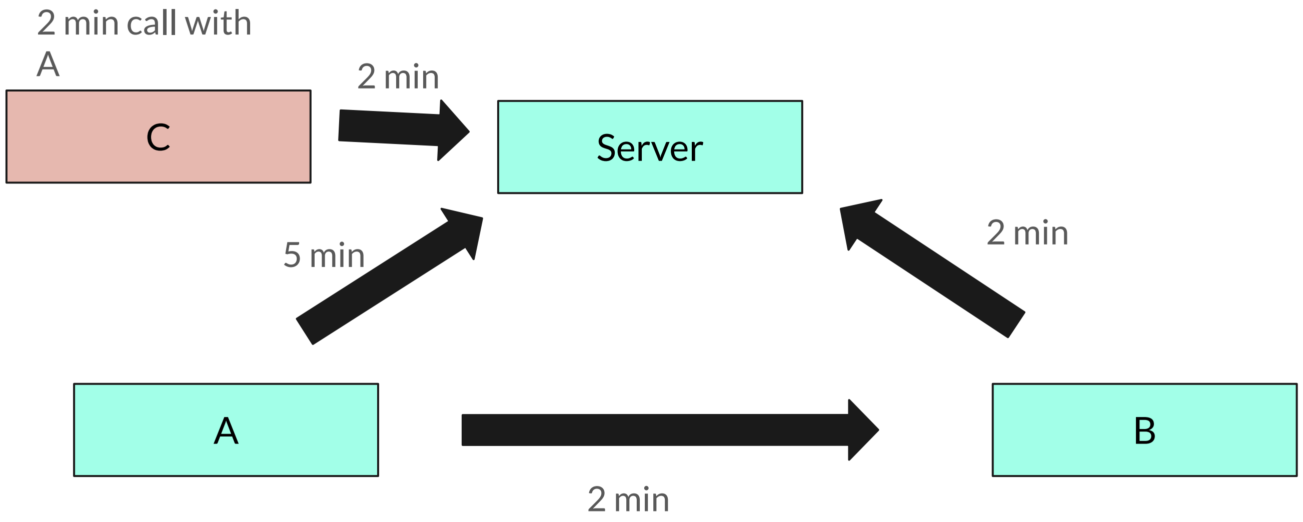
Reference Time:

10:00 - A sends req to server and B and external source C

10:02 - B receives req from A, sends req to server. C receives req from A

10:04 - B's request reaches server. C's request reaches server

10:05 - A's request reaches server

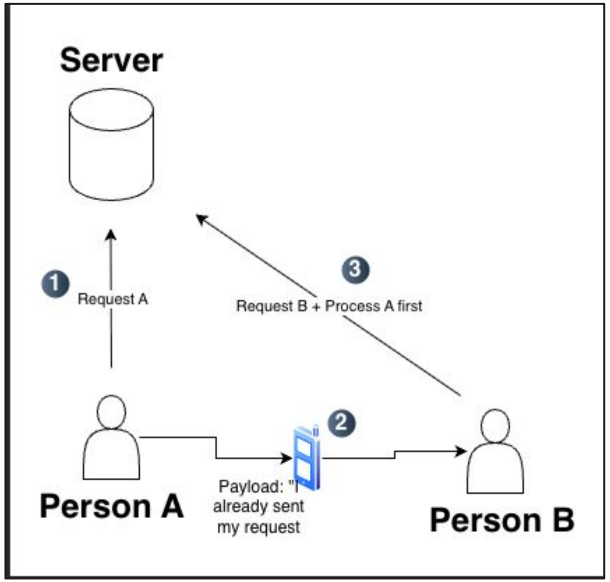


Causal Order	**ReqA -> ReqB** **ReqA -> ReqC**
Internal Order	**ReqB -> ReqA** **ReqC -> ReqA**

Avoid Anomalous Behavior

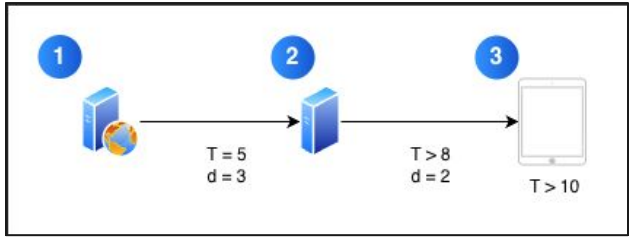
1. Explicitly carry casual information
   a. Attach prior event metadata to new requests
   b. Use causal ordering schemas (Lamport Timestamp, vector clocks, session chains)
   c. Systems enforce order based on user provided information

2. Synchronized Physical Clocks


Server
1
Request A
3
Request B + Process A first
2
Person A
Payload: "I already sent my request
Person B

Physical Clocks

- Purpose is to provide a real-world representation of time across distributed system
- **Strong Clock Condition**
  - If a -> b then C(a) < C(b)
- Implementation Rules
  - **PC1 Normal Rate:** clock rate must run forward and be close to a rate of 1.
    - Where [($dC_i(t) / dt$) > 0]
  - **PC2 Synchronization Rule:** difference between any two clocks must be sufficiently small
    - When a process **P** receives a message with timestamp **T** it must reset its clock forward to at least **T+d** where **d** is the minimum message delay



Bounding Synchronization Error

Formula: $\boldsymbol{\varepsilon = d(2\square T + \xi)}$ (maximum synchronization error the system can achieve)

Shows how far out of synchronization physical clocks will be from each other within the system

- **χ (Clock Accuracy) -** the quality of hardware / precision of clock
- **τ (Synchronization Interval):** time between synchronization messages / often drift is being reset
- **ξ (Unpredictable Delay):** variable delay in the network depending on its predictability
- **d (Graph Diameter):** size and topology of your distributed system
- **2 (Constant Value)**- accounts for worse case scenario where 2 clocks drift in opposite directions

Causal Consistency Requirement

Formula: $\varepsilon / (1 - \square) <= \mu$ (maximum clock difference <= minimum physical time of causality)

Real world condition for A -> B to imply C(A) < C(B)

- **ε - Synchronization error calculated from previous formula**
- **$(1 - \square)$ - lower limit of clock rate**
- **μ - minimum time for messages to travel between processes**

Physical Clocks - No External Causal Consistency

Reference Time:

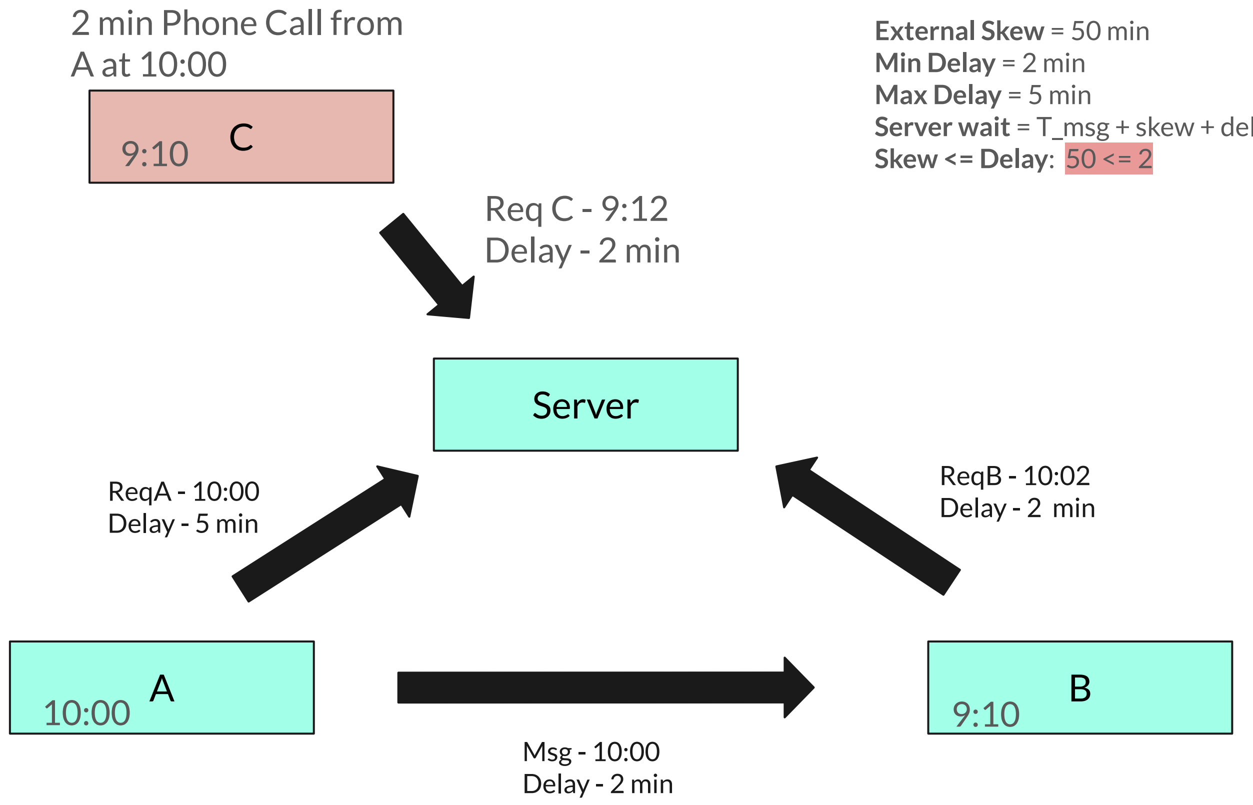
10:00 - A sends req to server and B

10:02 - B receives req from A, updates its own clock, sends req to server

10:04 - B's request reaches server

10:05 - A's request reaches server

10:30 - A's request reaches server



External Skew = 50 min
Min Delay = 2 min
Max Delay = 5 min
Server wait = T_msg + skew + delta delay
Skew <= Delay: 50 <= 2

Physical Clocks - With External Causal Consistency

Reference Time:

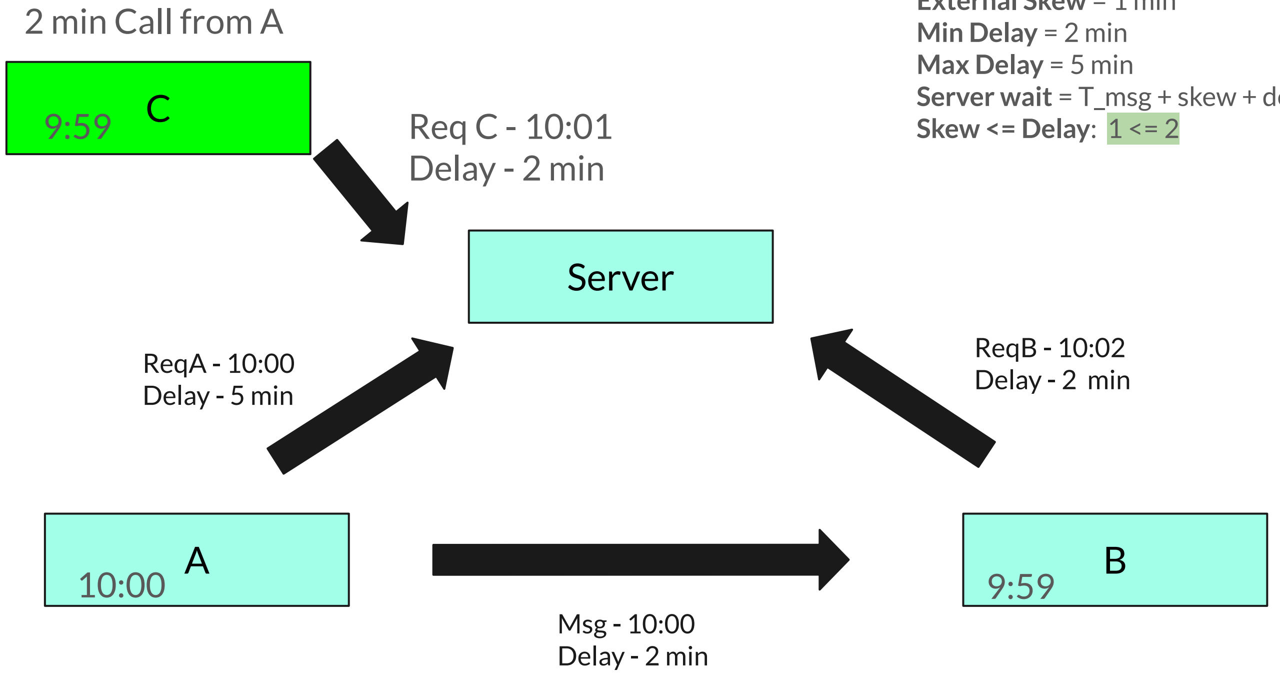
10:00 - A sends req to server and B

10:02 - B receives req from A, updates its own clock, sends req to server

10:04 - B's request reaches server

10:05 - A's request reaches server

External Skew = 1 min
Min Delay = 2 min
Max Delay = 5 min
Server wait = T_msg + skew + delta delay
Skew <= Delay: 1 <= 2



Conclusion

1. Defined the **“happens-before” relation** as a way to establish a **partial order of events** in distributed systems.

2. Proposed a method to **extend this partial order into a total order**, even if the total order is somewhat arbitrary.

3. Noted that enforcing total order can sometimes lead to **unnatural or misleading event sequences**.

4. Demonstrated how **synchronized physical clocks** can prevent such anomalies and presented an **algorithm to achieve this**.

Impossibility of distributed consensus with one faulty process.

Authors: Michael J. Fischer, Nancy A. Lynch, Michael S. Paterson

Asynchronous Model Definition

1. Processes interact through a message system which can delay messages arbitrarily long.

Asynchronous Model Definition

1. Processes interact through a message system which can delay messages arbitrarily long.
2. Message system is reliable, it will deliver the message correctly and exactly once.

Asynchronous Model Definition

1. Processes interact through a message system which can delay messages arbitrarily long.
2. Message system is reliable, it will deliver the message correctly and exactly once.
3. Processes do not have access to synchronized clocks.

Asynchronous Model Definition

1. Processes interact through a message system which can delay messages arbitrarily long.
2. Message system is reliable, it will deliver the message correctly and exactly once.
3. Processes do not have access to synchronized clocks.
4. Processes can not tell whether another process has died or is just running very slowly.

Note - we're not considering Byzantine failures, only whether a process dies unannounced.

Asynchronous Model Definition

1. Processes interact through a message system which can delay messages arbitrarily long.

2. Message system is reliable, it will deliver the message correctly and exactly once.

3. Processes do not have access to synchronized clocks.

4. Processes can not tell whether another process has died or is just running very slowly.

Note - we're not considering Byzantine failures, only whether a process dies unannounced.

Goal: To show that if one process dies at a specific point in time in an asynchronous system, then it's possible for the processes to not reach a consensus.

Consensus Protocol

A **consensus protocol** is an asynchronous system of N processors.

Consensus Protocol

A **consensus protocol** is an asynchronous system of N processors.

Each process **p** has an internal state consisting of:

- An input register (containing initial value)
- An output register (starting value is b and becomes either 0 or 1, decision)
- Unbounded internal memory

Consensus Protocol

A **consensus protocol** is an asynchronous system of N processors.

Each process **p** has an internal state consisting of:

- An input register (containing initial value)
- An output register (starting value is b and becomes either 0 or 1, decision)
- Unbounded internal memory

Initial states provide fixed starting values to the internal state except the output register.

Consensus Protocol

A **consensus protocol** is an asynchronous system of N processors.

Each process **p** has an internal state consisting of:

- An input register (containing initial value)
- An output register (starting value is b and becomes either 0 or 1, decision)
- Unbounded internal memory

Initial states provide fixed starting values to the internal state except the output register.

Processes send messages of the form **(p, m)** where **p** is destination process and **m** is the message.

Consensus Protocol

A **consensus protocol** is an asynchronous system of N processors.

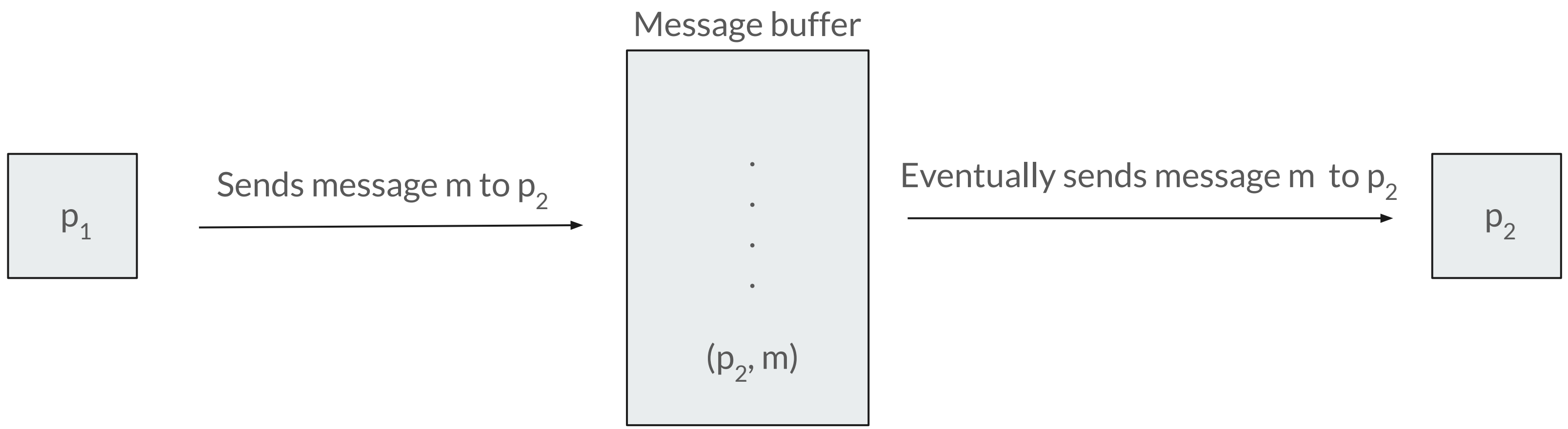
Each process **p** has an internal state consisting of:

- An input register (containing initial value)
- An output register (starting value is b and becomes either 0 or 1, decision)
- Unbounded internal memory

Initial states provide fixed starting values to the internal state except the output register.

Processes send messages of the form **(p, m)** where **p** is destination process and **m** is the message.

Messages are stored in a message buffer; processes request for a message from the message buffer but the system can delay it.


Message buffer
p_1
Sends message m to p_2
. . . .
(p_2, m)
Eventually sends message m to p_2
p_2

Consensus Protocol (cont.)

A *configuration* consists of the internal states of each process + content in the message buffer. The *initial configuration* is where each process is at its initial state and message buffer is empty.

Consensus Protocol (cont.)

A *configuration* consists of the internal states of each process + content in the message buffer. The *initial configuration* is where each process is at its initial state and message buffer is empty.

A *step* from one configuration to another consists of a primitive step done by some process **p**:

1. **p** requests any message sent to it from the message buffer

Consensus Protocol (cont.)

A *configuration* consists of the internal states of each process + content in the message buffer. The *initial configuration* is where each process is at its initial state and message buffer is empty.

A *step* from one configuration to another consists of a primitive step done by some process **p**:

1. **p** requests any message sent to it from the message buffer
2. Message buffer provides **m** (a valid message or $\varnothing$) to **p**.

Consensus Protocol (cont.)

A *configuration* consists of the internal states of each process + content in the message buffer. The *initial configuration* is where each process is at its initial state and message buffer is empty.

A *step* from one configuration to another consists of a primitive step done by some process **p**:

1. **p** requests any message sent to it from the message buffer
2. Message buffer provides **m** (a valid message or $\varnothing$) to **p**.
3. Then based off **m** and its current internal state, **p** will enter into a new state and then send a bunch of new messages to other processes which enter the message buffer (all deterministically).

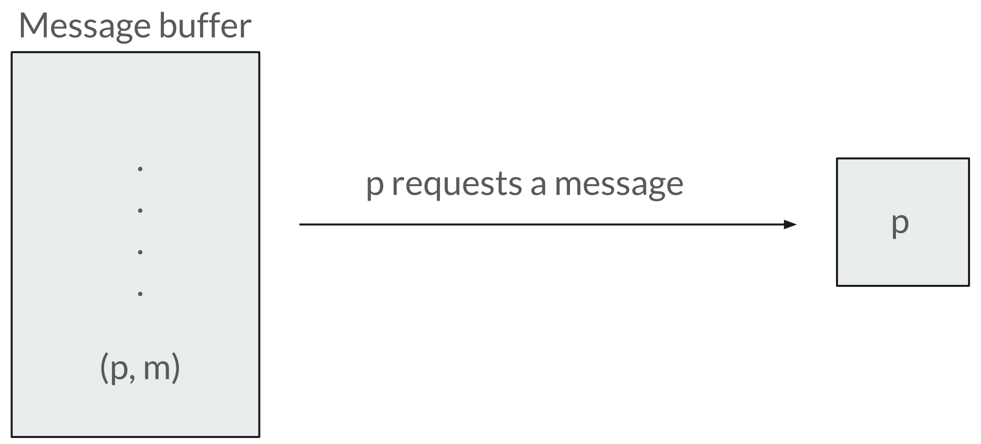
Consensus Protocol (cont.)

A *configuration* consists of the internal states of each process + content in the message buffer. The *initial configuration* is where each process is at its initial state and message buffer is empty.

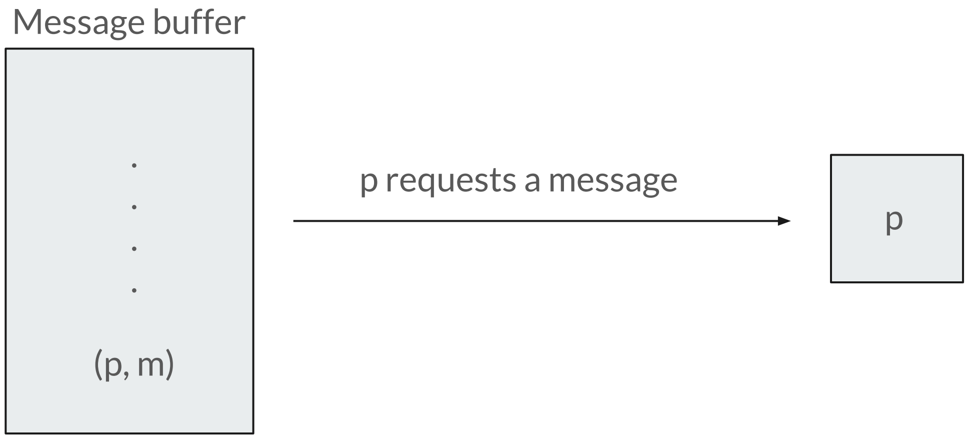
A *step* from one configuration to another consists of a primitive step done by some process **p**:

1. **p** requests any message sent to it from the message buffer
2. Message buffer provides **m** (a valid message or ∅) to **p**.
3. Then based off **m** and its current internal state, **p** will enter into a new state and then send a bunch of new messages to other processes which enter the message buffer (all deterministically).

We enter a new configuration and it’s purely determined by **e = (p, m)** which we call an *event*. A sequence of events is called a *run*.

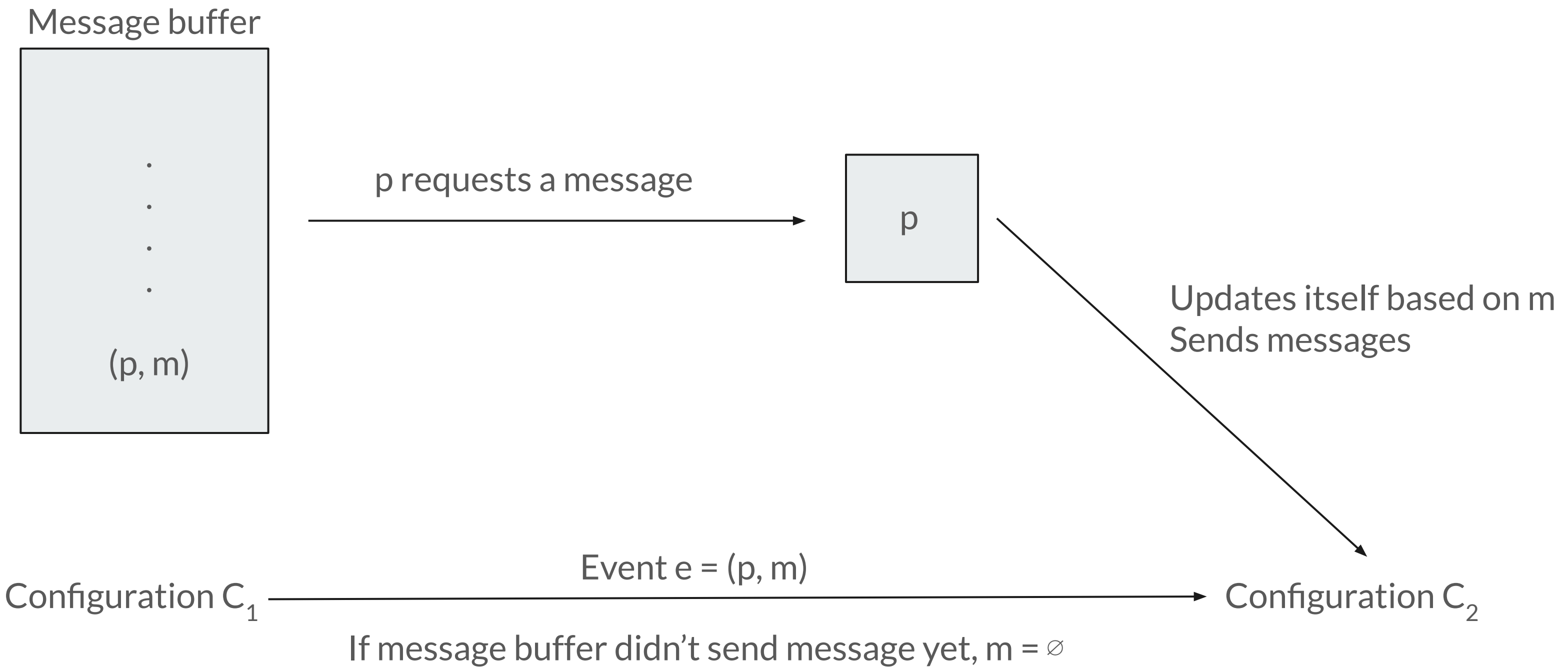


Configuration C_1

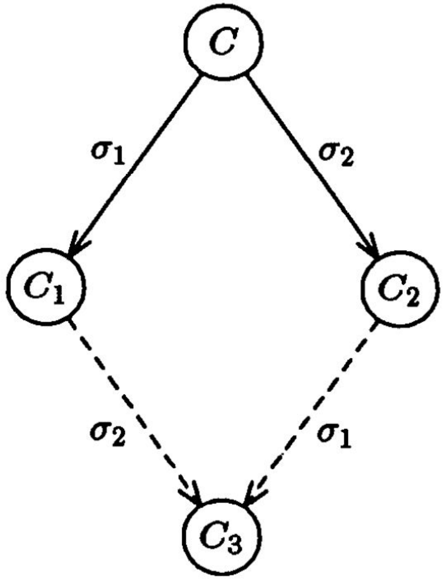


$$\text{Configuration } C_1 \xrightarrow{\text{Event } e = (p, m)}$$

If message buffer didn't send message yet, m = ∅


Message buffer
.
.
.
.
(p, m)
p requests a message
p
Updates itself based on m
Sends messages
Configuration C_1
Event e = (p, m)
Configuration C_2
If message buffer didn't send message yet, m = ∅

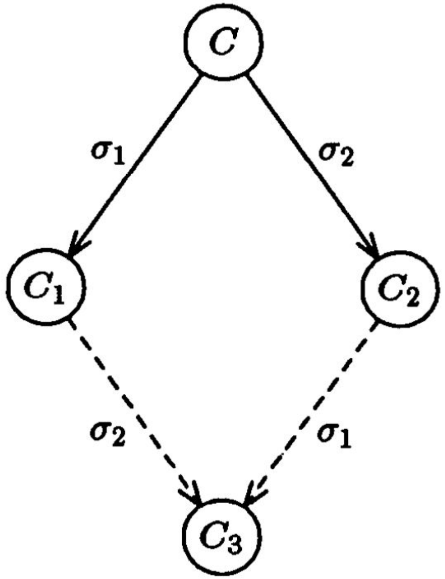
Consensus Protocol - commutative example


$$\begin{tikzcd} & C \arrow[dl, "\sigma_1"'] \arrow[dr, "\sigma_2"] & \\ C_1 \arrow[dr, dashed, "\sigma_2"'] & & C_2 \arrow[dl, dashed, "\sigma_1"] \\ & C_3 & \end{tikzcd}$$

Example of runs $\boldsymbol{\sigma}_1$ and $\boldsymbol{\sigma}_2$ being applied to a configuration.

If the events in these runs are disjoint in terms of the processes they're applied to, resulting configuration must be the same because processes act deterministically. "Commutative property"

Consensus Protocol - commutative example



Example of runs $\boldsymbol{\sigma}_1$ and $\boldsymbol{\sigma}_2$ being applied to a configuration.

If the events in these runs are disjoint in terms of the processes they're applied to, resulting configuration must be the same because processes act deterministically. "Commutative property"

We say that C is *accessible* if it is reachable by a run starting at some initial configuration.

Consensus Protocol - decision

Configuration has decision value *v* if some process outputs decision *v*. A run that leads to a decision being made is a *deciding* run.

Consensus Protocol - decision

Configuration has decision value *v* if some process outputs decision *v*. A run that leads to a decision being made is a *deciding* run.

A consensus protocol is *totally correct in spite of one fault* if:

1. No accessible configuration has more than one decision value (processes don't disagree)

Consensus Protocol - decision

Configuration has decision value *v* if some process outputs decision *v*. A run that leads to a decision being made is a *deciding* run.

A consensus protocol is *totally correct in spite of one fault* if:

1. No accessible configuration has more than one decision value (processes don’t disagree)
2. There exists accessible configurations with decision value 0 and 1 (choice in agreement)

Consensus Protocol - decision

Configuration has decision value *v* if some process outputs decision *v*. A run that leads to a decision being made is a *deciding* run.

A consensus protocol is *totally correct in spite of one fault* if:

1. No accessible configuration has more than one decision value (processes don't disagree)
2. There exists accessible configurations with decision value 0 and 1 (choice in agreement)
3. All processes take infinitely many steps except the faulty process

Consensus Protocol - decision

Configuration has decision value *v* if some process outputs decision *v*. A run that leads to a decision being made is a *deciding* run.

A consensus protocol is *totally correct in spite of one fault* if:

1. No accessible configuration has more than one decision value (processes don't disagree)
2. There exists accessible configurations with decision value 0 and 1 (choice in agreement)
3. All processes take infinitely many steps except the faulty process
4. All messages sent to nonfaulty processes are eventually received (run is *admissible*)

Consensus Protocol - decision

Configuration has decision value *v* if some process outputs decision *v*. A run that leads to a decision being made is a *deciding* run.

A consensus protocol is *totally correct in spite of one fault* if:

1. No accessible configuration has more than one decision value (processes don't disagree)
2. There exists accessible configurations with decision value 0 and 1 (choice in agreement)
3. All processes take infinitely many steps except the faulty process
4. All messages sent to nonfaulty processes are eventually received (run is *admissible*)

Note - first two conditions are called being *partially correct*

Consensus Protocol - decision

Configuration has decision value *v* if some process outputs decision *v*. A run that leads to a decision being made is a *deciding* run.

A consensus protocol is *totally correct in spite of one fault* if:

1. No accessible configuration has more than one decision value (processes don't disagree)
2. There exists accessible configurations with decision value 0 and 1 (choice in agreement)
3. All processes take infinitely many steps except the faulty process
4. All messages sent to nonfaulty processes are eventually received (run is *admissible*)

Note - first two conditions are called being *partially correct*

Goal: every partially correct protocol has an admissible run that is not a deciding run

Admissible run:

1. All processes take infinitely many steps except the faulty process.
2. All messages sent to nonfaulty processes are eventually received.

Theorem 1. No consensus protocol is totally correct in spite of one fault.

Admissible run:

1. All processes take infinitely many steps except the faulty process.
2. All messages sent to nonfaulty processes are eventually received.

Theorem 1. No consensus protocol is totally correct in spite of one fault.

Proof. Assume to the contrary that P is a consensus protocol that is totally correct in spite of one fault. We prove a sequence of lemmas which eventually lead to a contradiction.

Admissible run:
1. All processes take infinitely many steps except the faulty process.
2. All messages sent to nonfaulty processes are eventually received.

Theorem 1. No consensus protocol is totally correct in spite of one fault.

Proof. Assume to the contrary that P is a consensus protocol that is totally correct in spite of one fault. We prove a sequence of lemmas which eventually lead to a contradiction.

The premise of the proof is to show that the protocol can be kept forever indecisive even for an admissible run.

Admissible run:

1. All processes take infinitely many steps except the faulty process.
2. All messages sent to nonfaulty processes are eventually received.

Theorem 1. No consensus protocol is totally correct in spite of one fault.

Proof. Assume to the contrary that P is a consensus protocol that is totally correct in spite of one fault. We prove a sequence of lemmas which eventually lead to a contradiction.

The premise of the proof is to show that the protocol can be kept forever indecisive even for an admissible run.

For there to exist an indecisive run, there needs to exist some initial configuration where the outcome isn't already decided. Then we need to demonstrate that there is an admissible run which avoids ever committing to a decision.

First we'll show that there must exist some initial configuration where the output is not predetermined.

Lemma 2. P has a bivalent initial configuration

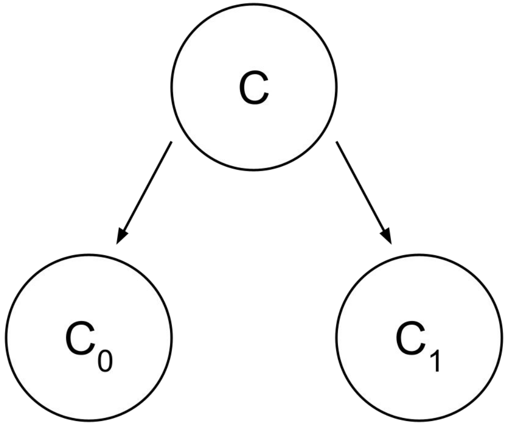
First we'll show that there must exist some initial configurations where the output is not predetermined.

Configuration C is bivalent if it can reach configurations with decision values of both 0 and 1

Lemma 2. P has a bivalent initial configuration

First we'll show that there must exist some initial configurations where the output is not predetermined.

Configuration C is bivalent if it can reach configurations with decision values of both 0 and 1



Subscripts indicate decision value

Lemma 2. P has a bivalent initial configuration

Proof. Assume to the contrary that there is no bivalent initial configuration.

Lemma 2. P has a bivalent initial configuration

Proof. Assume to the contrary that there is no bivalent initial configuration.

Then P must have 0-valent and 1-valent initial configurations by the assumed partial correctness.

Recall the definition of partial correctness:

1. No accessible configuration has more than one decision value
2. <u>There exists accessible configurations with decision value 0 and 1</u>

Lemma 2. P has a bivalent initial configuration

Proof. Assume to the contrary that there is no bivalent initial configuration.

Then P must have 0-valent and 1-valent initial configurations by the assumed partial correctness.

Recall the definition of partial correctness:

1. No accessible configuration has more than one decision value
2. <u>There exists accessible configurations with decision value 0 and 1</u>

We will call two initial configurations *adjacent* if they differ only in the initial value x_p of a single process p.

Lemma 2. P has a bivalent initial configuration

Proof. Assume to the contrary that there is no bivalent initial configuration.

Then P must have 0-valent and 1-valent initial configurations by the assumed partial correctness.

Recall the definition of partial correctness:

1. No accessible configuration has more than one decision value
2. There exists accessible configurations with decision value 0 and 1

We will call two initial configurations *adjacent* if they differ only in the initial value x_p of a single process p.

Examples:

$$\texttt{Initial Config 1} = \{[x_0=0],[x_1=0],[x_2=0],[x_3=0],[x_4=0]\}$$
$$\texttt{Initial Config 2} = \{[x_0=1],[x_1=0],[x_2=0],[x_3=0],[x_4=0]\}$$

Lemma 2. P has a bivalent initial configuration

There must exist a 0-valent initial configuration C_0 adjacent to a 1-valent initial configuration C_1.

Lemma 2. P has a bivalent initial configuration

There must exist a 0-valent initial configuration C_0 adjacent to a 1-valent initial configuration C_1.

Why?
Let an initial configuration where all initial n values are set to 0 be 0-valent.

```
    {[x_0=0],[x_1=0],[x_2=0], ... ,[x_n=0]}
```

Let an initial configuration where all initial n values are set to 1 be 1-valent.

```
    {[x_0=1],[x_1=1],[x_2=1], ... ,[x_n=1]}
```

$C^{(k)}$ represents an initial configuration where the first k initial values are set to 1, the rest to 0.

Lemma 2. P has a bivalent initial configuration

There must exist a 0-valent initial configuration C_0 adjacent to a 1-valent initial configuration C_1.

Why?
Let an initial configuration where all initial n values are set to 0 be 0-valent. -> $C^{(0)}$

```
    {[x0=0],[x1=0],[x2=0], ... ,[xn=0]}
```

Let an initial configuration where all initial n values are set to 1 be 1-valent. -> $C^{(n)}$

```
    {[x0=1],[x1=1],[x2=1], ... ,[xn=1]}
```

$C^{(k)}$ represents an initial configuration where the first k initial values are set to 1, the rest to 0.

Lemma 2. P has a bivalent initial configuration

There must exist a 0-valent initial configuration C_0 adjacent to a 1-valent initial configuration C_1.
$C^{(k)}$ represents an initial configuration where the first k initial values are set to 1, the rest to 0.

This is a list of adjacent initial configurations since they differ by 1 initial value in each step:

$$C^{(0)}, C^{(1)}, C^{(2)}, C^{(3)}, \dots , C^{(n-1)}, C^{(n)}$$

Lemma 2. P has a bivalent initial configuration

There must exist a 0-valent initial configuration C_0 adjacent to a 1-valent initial configuration C_1. $C^{(k)}$ represents an initial configuration where the first k initial values are set to 1, the rest to 0.

This is a list of adjacent initial configurations since they differ by 1 initial value in each step:

$$C^{(0)}, C^{(1)}, C^{(2)}, C^{(3)}, \dots , C^{(n-1)}, C^{(n)}$$

Since $C^{(0)}$ is 0-valent, and $C^{(n)}$ is 1-valent, and we are assuming there are no bivalent initial configurations, there must be some boundary $C^{(k)}$, $C^{(k+1)}$ where $C^{(k)}$ is 0-valent and $C^{(k+1)}$ is 1-valent.

Lemma 2. P has a bivalent initial configuration

There must exist a 0-valent initial configuration C_0 adjacent to a 1-valent initial configuration C_1.
$C^{(k)}$ represents an initial configuration where the first k initial values are set to 1, the rest to 0.

This is a list of adjacent initial configurations since they differ by 1 initial value in each step:

$$C^{(0)}, C^{(1)}, C^{(2)}, C^{(3)}, \dots , C^{(n-1)}, C^{(n)}$$

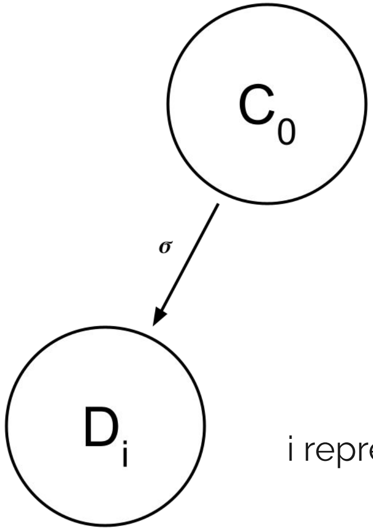
Since $C^{(0)}$ is 0-valent, and $C^{(n)}$ is 1-valent, and we are assuming there are no bivalent initial configurations, there must be some boundary $C^{(k)}$, $C^{(k+1)}$ where $C^{(k)}$ is 0-valent and $C^{(k+1)}$ is 1-valent.

So we've proven that C_0 and C_1 exist. Let p be the process in whose initial value they differ.

Lemma 2. P has a bivalent initial configuration

Consider an admissible deciding run from C_0 *where process p takes no steps*, and let σ be the associated schedule.

<u>Note:</u>
C_0 and C_1 differ only in the initial value x_p of a single process p.

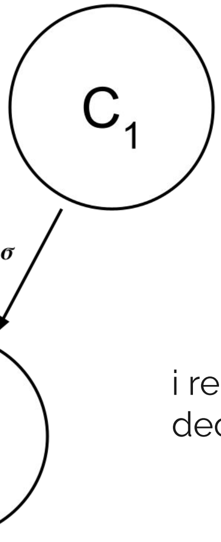


C_0 is 0-valent

i represents a decision value, either 0 or 1

Lemma 2. P has a bivalent initial configuration

Then σ can also be applied to C_1, and they result in the same decision.



C_1 is 1-valent

Note:
C_0 and C_1 differ only in the initial value x_p of a single process p.

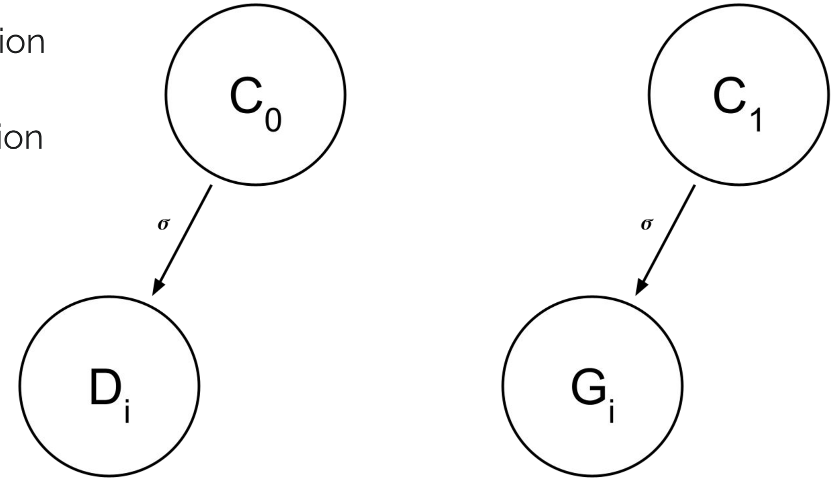
i represents the *same* decision value as previously

Lemma 2. P has a bivalent initial configuration

For them to reach the same decision, one of the initial configurations *must* be bivalent. Whichever is bivalent, our assumption that no bivalent configuration exists has been disproven.□

Let i = 0, then C_1 is a bivalent initial configuration

Let i = 1, then C_0 is a bivalent initial configuration



Intermission

We've proven **Lemma 2**. P has a bivalent initial configuration

For there to exist an indecisive run, there needs to exist some initial configuration where the outcome isn't already decided. Then we need to demonstrate that there is an admissible run which avoids ever committing to a decision.

The red portion is done, but we need more to figure out the 2nd part.
For that, we need **Lemma 3**.

Lemma 3.

Let C be a bivalent configuration of P

Lemma 3.

Let C be a bivalent configuration of P
Let e = (p, m) be an event that is applicable to C

Lemma 3.

Let C be a bivalent configuration of P

Let e = (p, m) be an event that is applicable to C

Let $\mathbb{C}$ be the set of configurations reachable from C without applying e

Lemma 3.

Let C be a bivalent configuration of P
Let e = (p, m) be an event that is applicable to C
Let $\mathbb{C}$ be the set of configurations reachable from C without applying e
Let $\mathbb{D} = e(\mathbb{C}) = \{e(E) \mid E \in \mathbb{C}$ and e is applicable to E$\}$

Lemma 3.

Let C be a bivalent configuration of P
Let e = (p, m) be an event that is applicable to C
Let $\mathbb{C}$ be the set of configurations reachable from C without applying e
Let $\mathbb{D} = e(\mathbb{C}) = \{e(E) \mid E \in \mathbb{C}$ and e is applicable to E$\}$
Then $\mathbb{D}$ contains a bivalent configuration

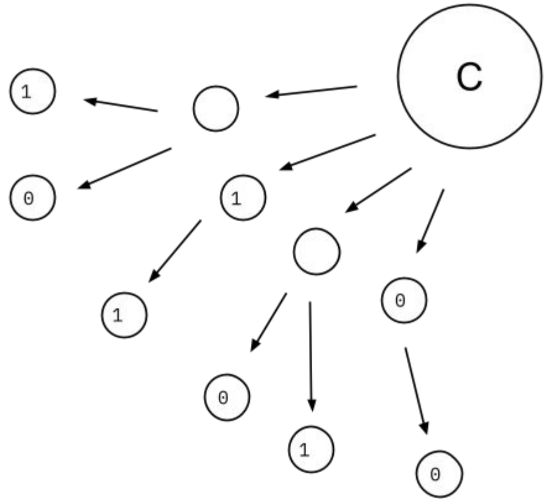
Let C be a bivalent configuration of P

Let e = (p, m) be an event that is applicable to C

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

Let 𝔻 = e(ℂ) = {e(E) | E ∈ ℂ and e is applicable to E}

Then 𝔻 contains a bivalent configuration



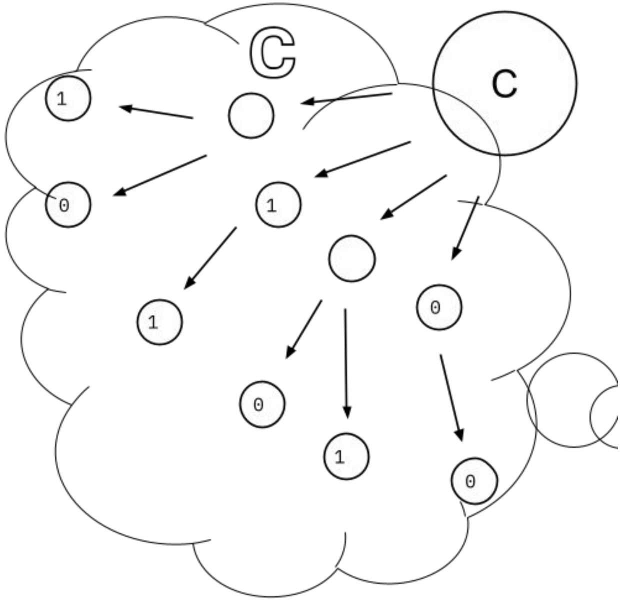
Let C be a bivalent configuration of P

Let e = (p, m) be an event that is applicable to C

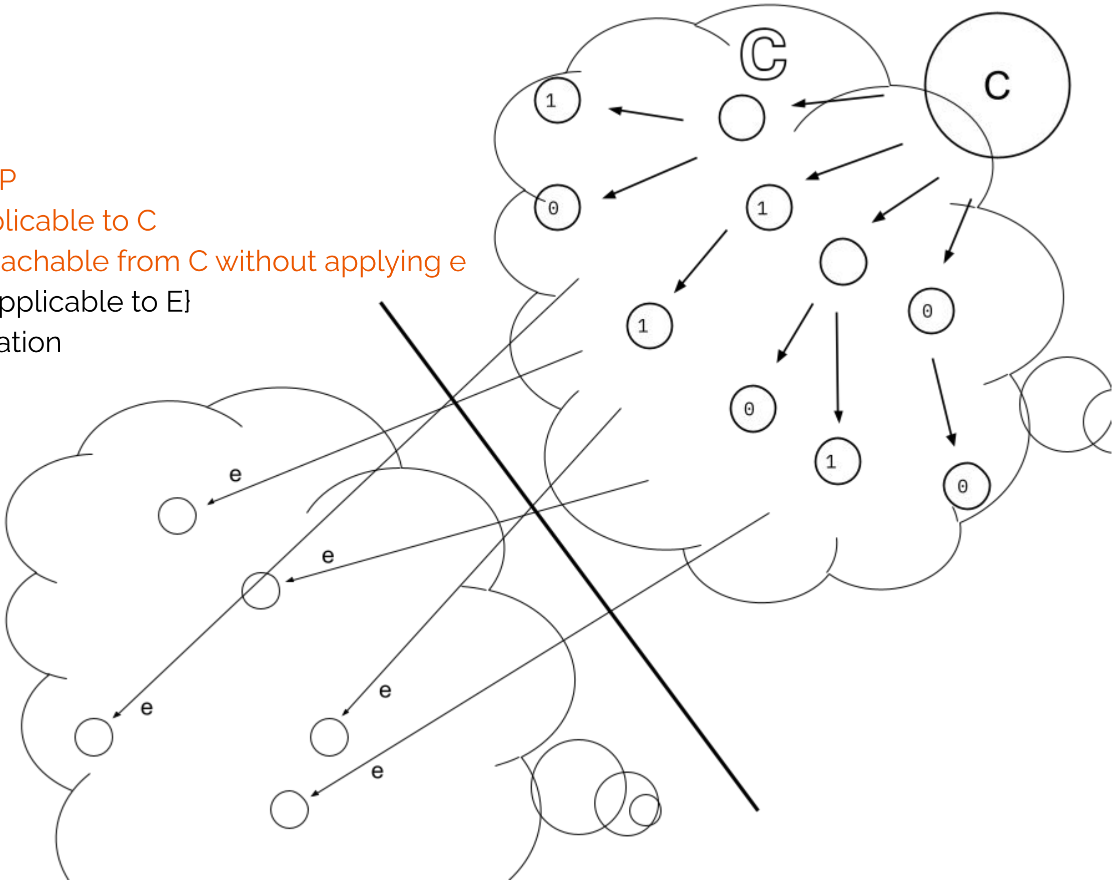
Let ℂ be the set of configurations reachable from C without applying e

Let 𝔻 = e(ℂ) = {e(E) | E ∈ ℂ and e is applicable to E}

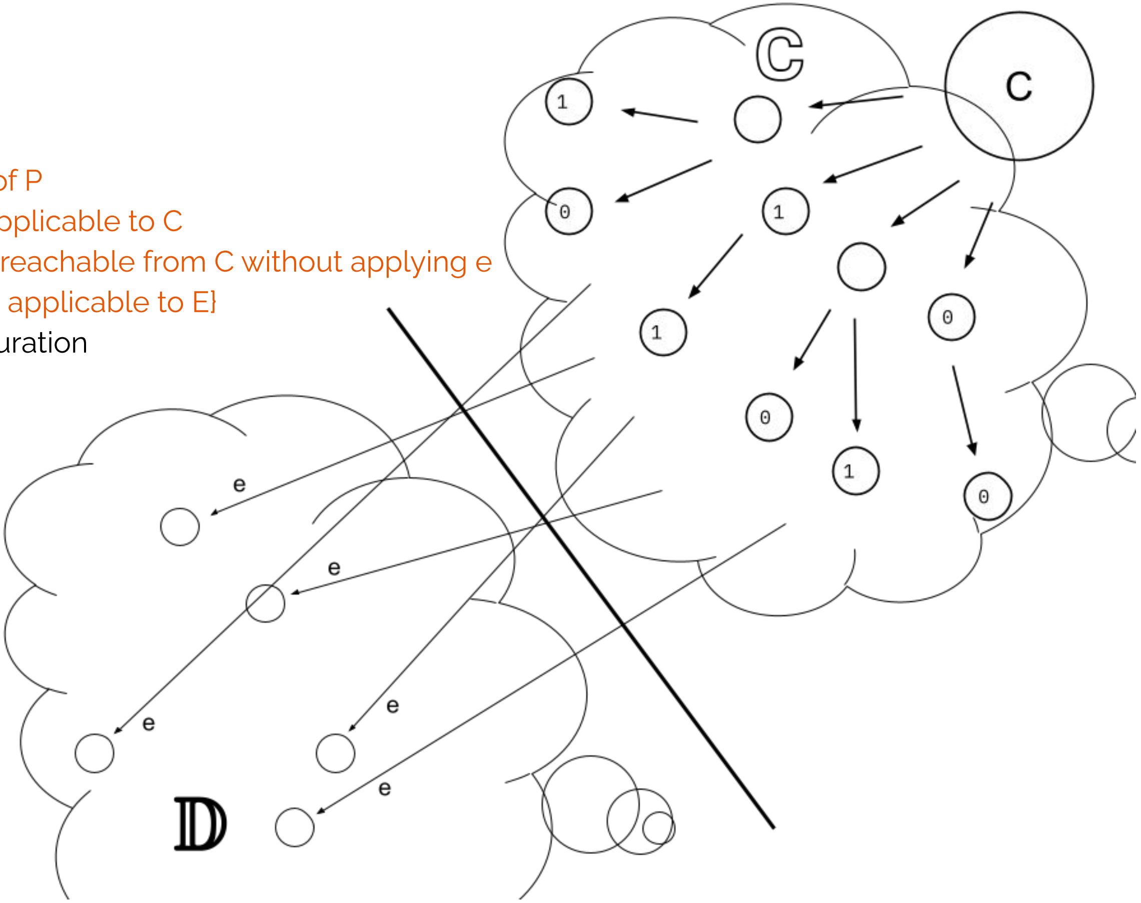
Then 𝔻 contains a bivalent configuration



Let C be a bivalent configuration of P
Let e = (p, m) be an event that is applicable to C
Let ℂ be the set of configurations reachable from C without applying e
Let 𝔻 = e(ℂ) = {e(E) | E ∈ ℂ and e is applicable to E}
Then 𝔻 contains a bivalent configuration



Let C be a bivalent configuration of P
Let e = (p, m) be an event that is applicable to C
Let ℂ be the set of configurations reachable from C without applying e
Let 𝔻 = e(ℂ) = {e(E) | E ∈ ℂ and e is applicable to E}
Then 𝔻 contains a bivalent configuration



Lemma 3.

Proof.
The event e is applicable to every E $\in \mathbb{C}$, since $\mathbb{C}$ was the set of configs reachable from C without applying e, and messages can be delayed arbitrarily. In other words, e can be applied at any point during a run.

Lemma 3.

Proof.
The event e is applicable to every E $\in \mathbb{C}$, since $\mathbb{C}$ was the set of configs reachable from C without applying e, and messages can be delayed arbitrarily. In other words, e can be applied at any point during a run.

Recall the last line of Lemma 3: “Then $\mathbb{D}$ contains a bivalent configuration”

Lemma 3.

Proof.
The event e is applicable to every E $\in \mathbb{C}$, since $\mathbb{C}$ was the set of configs reachable from C without applying e, and messages can be delayed arbitrarily. In other words, e can be applied at any point during a run.

Recall the last line of Lemma 3: “Then $\mathbb{D}$ contains a bivalent configuration”

Now assume to the contrary that $\mathbb{D}$ contains no bivalent configurations.
So every configuration D$\in\mathbb{D}$ is univalent. We proceed to find a contradiction.

Lemma 3.

Next we define an i-valent configuration E_i reachable from C, i = 0, 1.
E_i exists because C is bivalent.

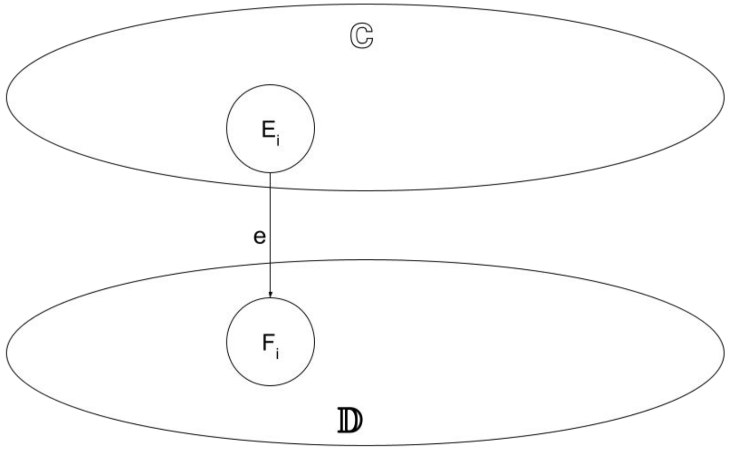
Lemma 3.

Next we define an i-valent configuration E_i reachable from C, i = 0, 1.
E_i exists because C is bivalent.
This E_i might be in $\mathbb{C}$, or it might not. We'll deal with both possibilities.

Lemma 3.

Next we define an i-valent configuration E_i reachable from C, i = 0, 1.
E_i exists because C is bivalent.
This E_i might be in $\mathbb{C}$, or it might not. We'll deal with both possibilities.

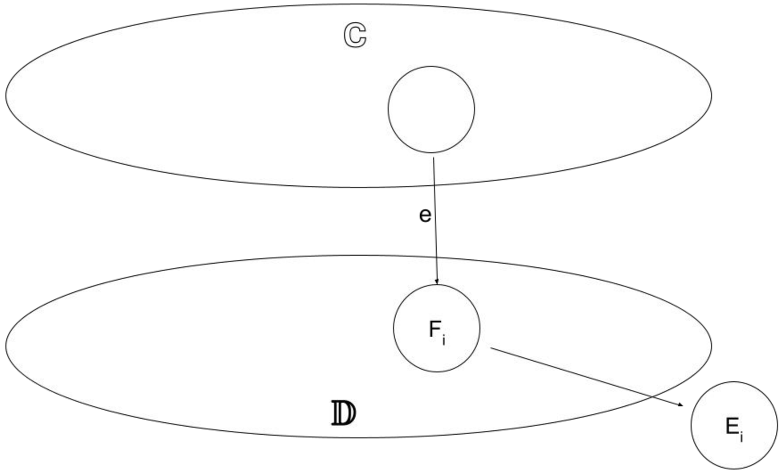
If $E_i \in \mathbb{C}$, let $F_i = e(E_i) \in \mathbb{D}$



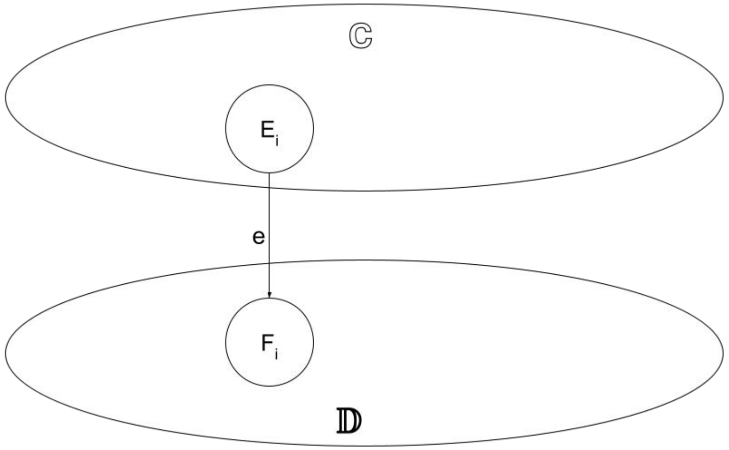
Lemma 3.

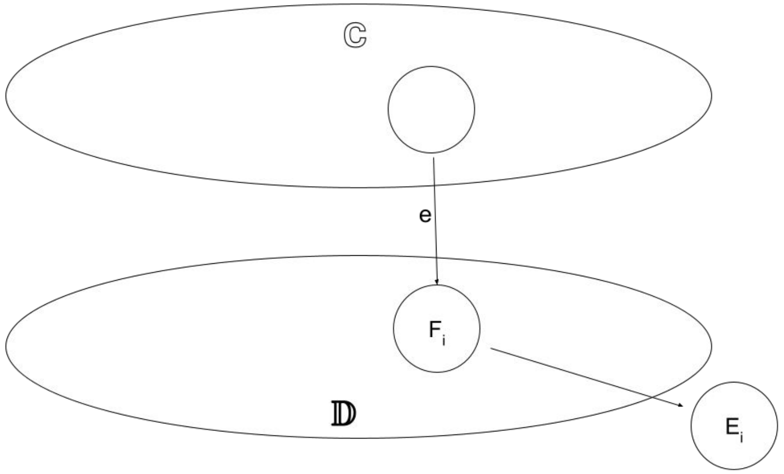
Next we define an i-valent configuration E_i reachable from C, i = 0, 1.
E_i exists because C is bivalent.
This E_i might be in $\mathbb{C}$, or it might not. We'll deal with both possibilities.

If $E_i \notin \mathbb{C}$, then e was applied
In reaching E_i. So there's an
$F_i \in \mathbb{D}$ from which E_i is
reachable



In both cases, $F_i \in \mathbb{D}$ is i-valent, since E_i and F_i are reachable from each other, both E_0 and E_1 exist, and we assumed that "$\mathbb{D}$ contains no bivalent configurations."
<u>So $\mathbb{D}$ contains both 0-valent and 1-valent configurations.</u>

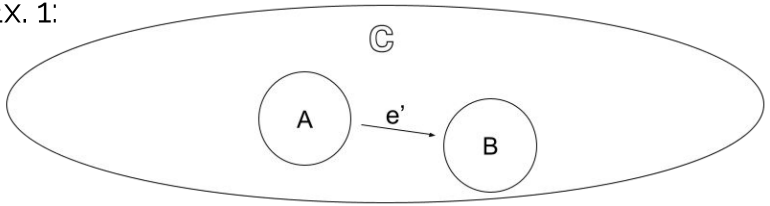




We call two configurations *neighbors* if one results from the other in a single step.

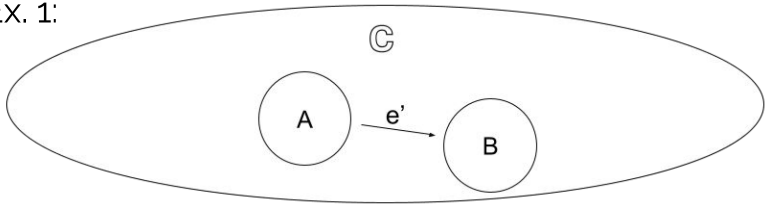
We call two configurations *neighbors* if one results from the other in a single step.

Ex. 1:

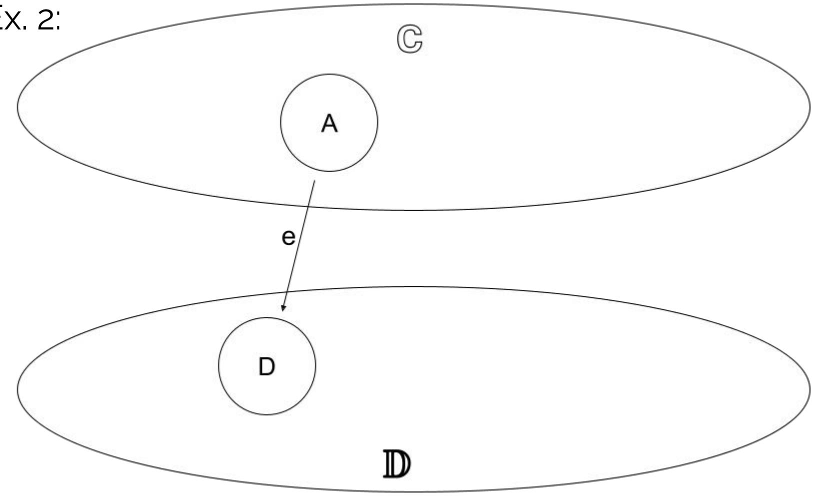


We call two configurations *neighbors* if one results from the other in a single step.

Ex. 1:

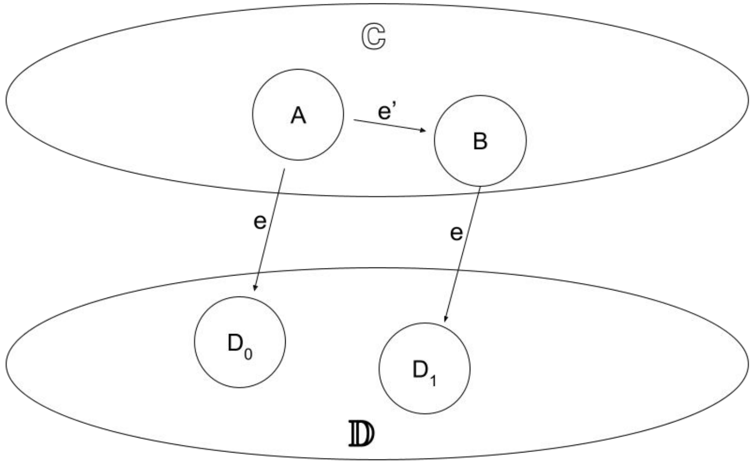


Ex. 2:



We call two configurations *neighbors* if one results from the other in a single step.
Since we proved $\mathbb{D}$ contains both 0-valent and 1-valent configurations, we can say the following:

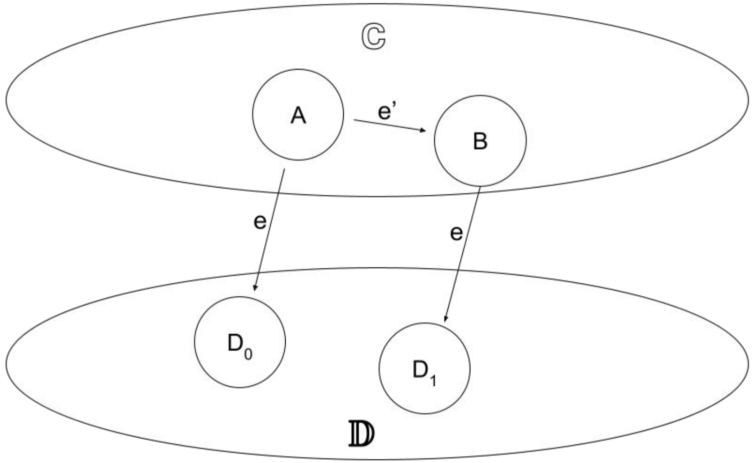
There exist neighbors A, B $\in \mathbb{C}$ such that $D_0 \in \mathbb{D} = e(A)$ and $D_1 \in \mathbb{D} = e(B)$, subscripts denote valency



We call two configurations *neighbors* if one results from the other in a single step.
Since we proved $\mathbb{D}$ contains both 0-valent and 1-valent configurations, we can say the following:

There exist neighbors A, B $\in \mathbb{C}$ such that $D_0 \in \mathbb{D} = e(A)$ and $D_1 \in \mathbb{D} = e(B)$, subscripts denote valency

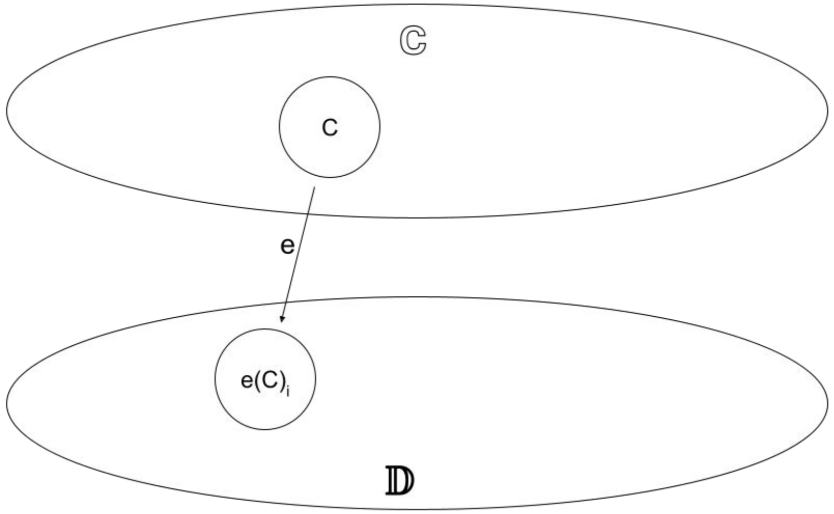
Why?



Recall:

1. Configuration C can reach any member of $\mathbb{C}$ without applying e.
2. e is applicable to every configuration in $\mathbb{C}$, and the result is some D $\in \mathbb{D}$.
3. $\mathbb{D}$ contains only univalent configurations, both 0-valent and 1-valent.

e(C) is some i-valent configuration $\in \mathbb{D}$.

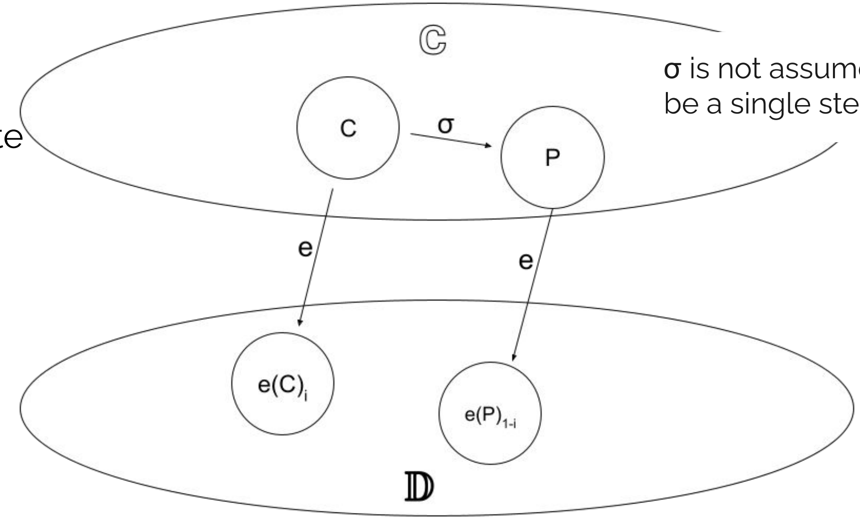


Recall:

1. Configuration C can reach any member of ℂ without applying e.
2. e is applicable to every configuration in ℂ, and the result is some D ∈ 𝔻.
3. 𝔻 contains only univalent configurations, both 0-valent and 1-valent.

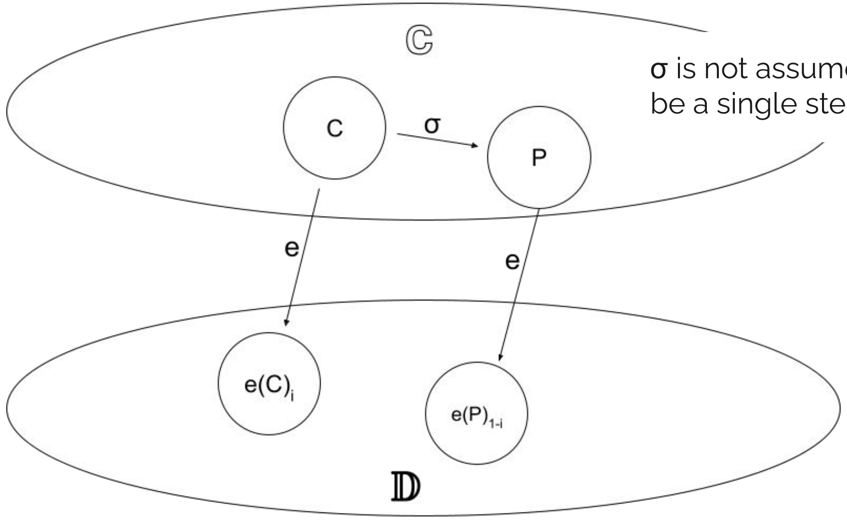
e(C) is some i-valent configuration ∈ **𝔻.**

Let P ∈ ℂ where e(P) ∈ 𝔻 has the opposite valency of e(C) (ie. e(P) is (1 - i)-valent).



σ is not assumed to be a single step

$C^{(k)}$ represents a configuration along the run from C to P, where each value k is a single step.

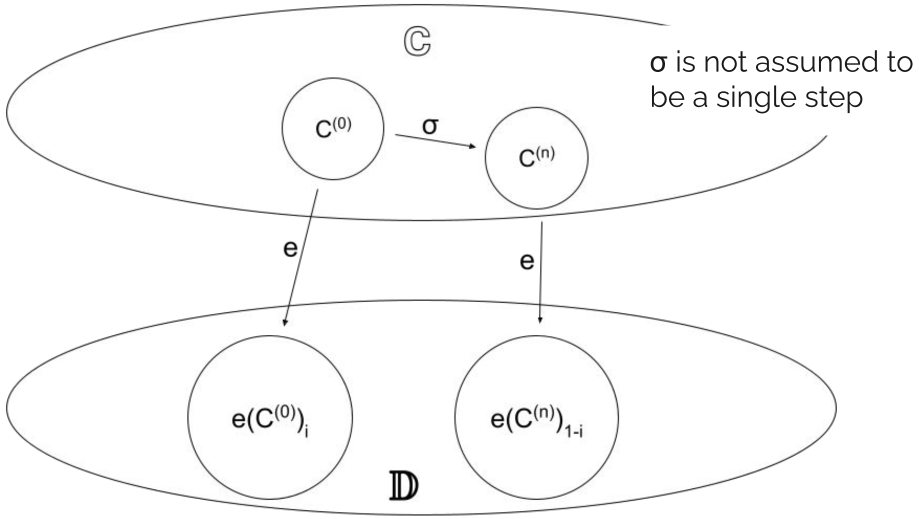


σ is not assumed to be a single step

$C^{(k)}$ represents a configuration along the run from C to P, where each value k is a single step.

Using this notation: $C^{(o)}$ = C,

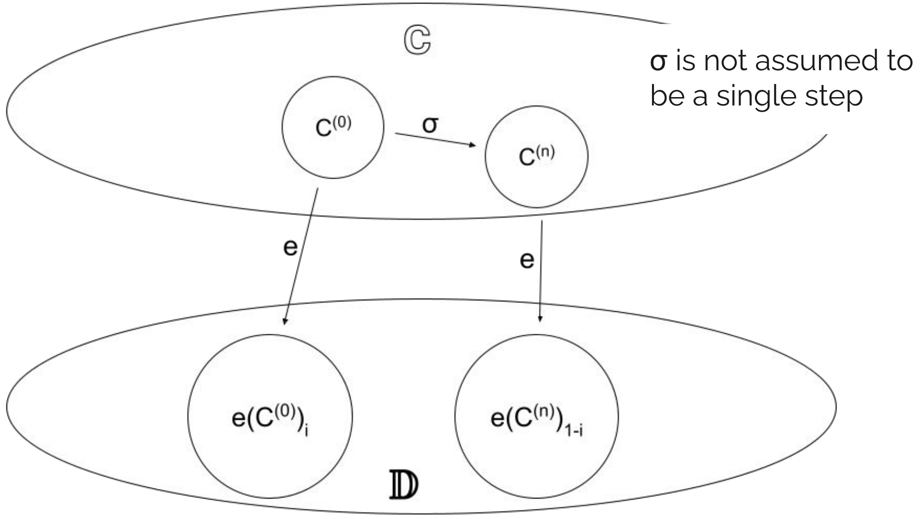
$C^{(n)}$ = P



$C^{(k)}$ represents a configuration along the run from $C^{(0)}$ to $C^{(n)}$, where each value k is a single step.

This is a list of configurations along the run from $C^{(0)}$ to $C^{(n)}$ where <u>adjacent configurations are neighbors</u> since they differ by a single step in the run:

$$C^{(0)}, C^{(1)}, C^{(2)}, C^{(3)}, \ldots, C^{(n-1)}, C^{(n)}$$

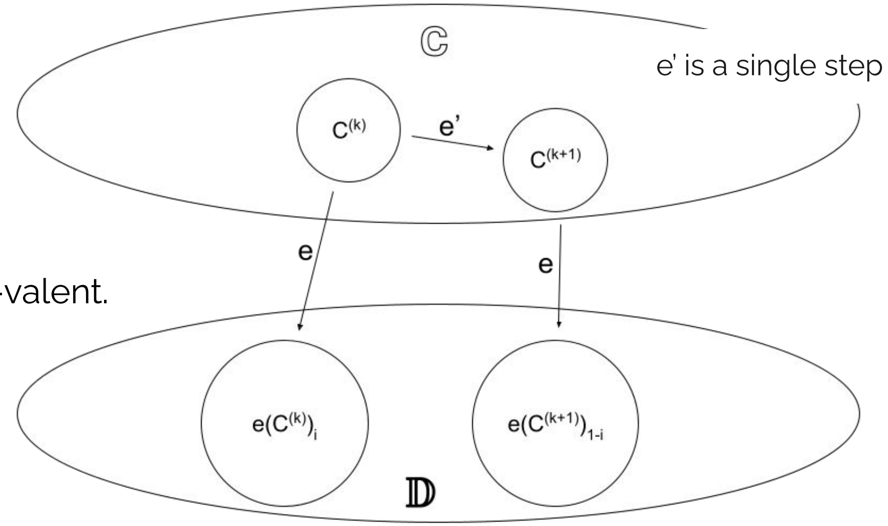


$C^{(k)}$ represents a configuration along the run from $C^{(0)}$ to $C^{(n)}$, where each value k is a single step.

This is a list of configurations along the run from $C^{(0)}$ to $C^{(n)}$, where adjacent configurations are neighbors since they differ by a single step in the run:

$$C^{(0)}, C^{(1)}, C^{(2)}, C^{(3)}, \dots, C^{(n-1)}, C^{(n)}$$

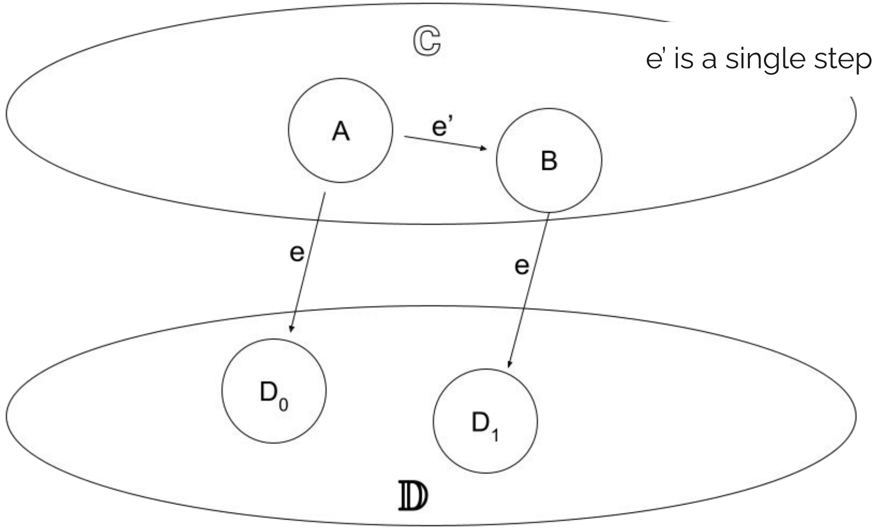
Since $e(C^{(0)})$ is i-valent, and $e(C^{(n)})$ is (1 - i)-valent, and we are assuming there are no bivalent configurations in $\mathbb{D}$, there must be some boundary $C^{(k)}$, $C^{(k+1)}$ where $e(C^{(k)})$ is i-valent and $e(C^{(k+1)})$ is (1 - i)-valent.



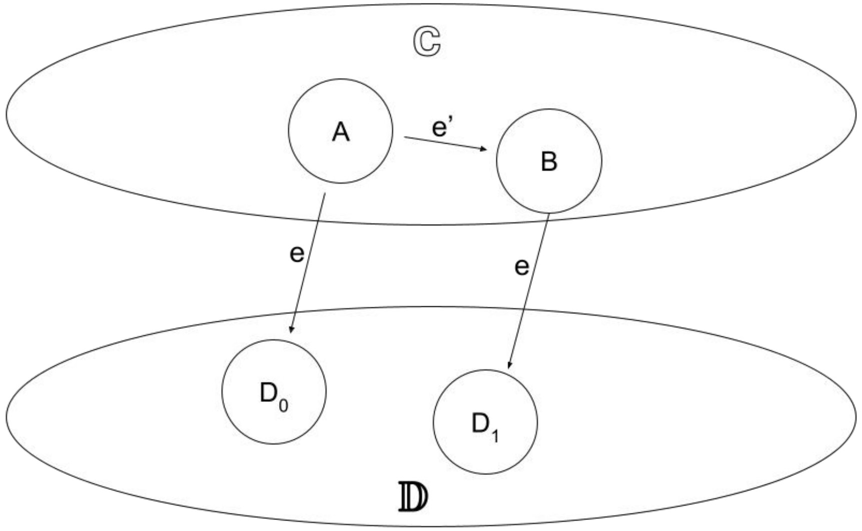
Since $e(C^{(0)})$ is i-valent, and $e(C^{(n)})$ is (1 - i)-valent, and we are assuming there are no bivalent configurations in $\mathbb{D}$, there must be some boundary $C^{(k)}$, $C^{(k+1)}$ where $e(C^{(k)})$ is i-valent and $e(C^{(k+1)})$ is (1 - i)-valent.

Let $A = C^{(k)}$
$B = C^{(k+1)}$
$D_0 = D_i = e(C^{(k)})$
$D_1 = D_{i-1} = e(C^{(k+1)})$
e' = (p', m') be the event taking A to B

All for ease of presentation.



Since A and B are neighbors, there must be an event e' = (p', m') which allows one to reach the other in a single step. B = e'(A)

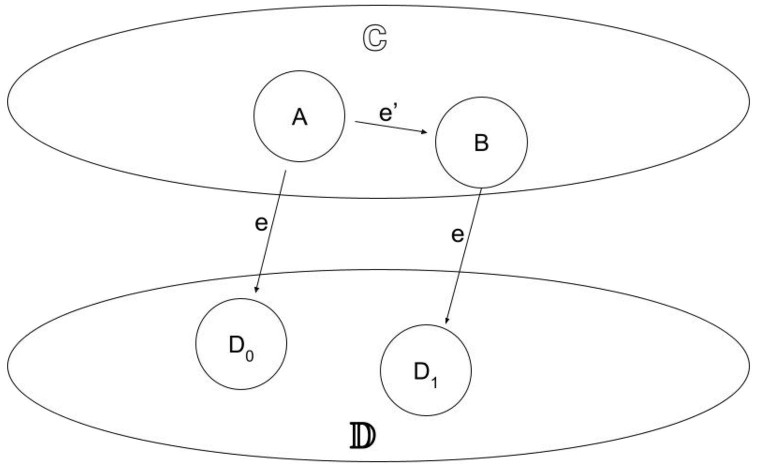


Recall from definition of Lemma 3: “Let e = (p, m) be an event that is applicable to C”,
And from the previous slide: “there must be an event e’ = (p’, m’)”

Recall from definition of Lemma 3: “Let $e = (p, m)$ be an event that is applicable to C”,
And from the previous slide: “there must be an event $e' = (p', m')$”

So there are two cases to concern ourselves with. $p' \neq p$ and $p' = p$.

Case 1. p’ ≠ p
Recall: disjoint schedules can be run in any order, leading to the same configuration.
The events e and e’ are disjoint since p’ ≠ p

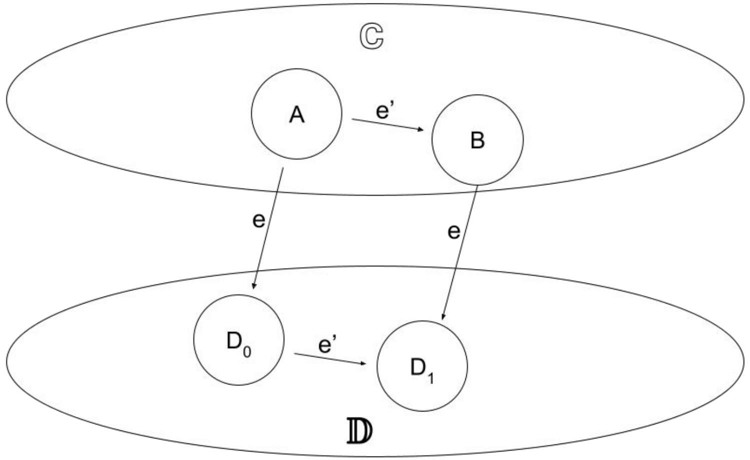


Case 1. p' ≠ p

Recall: disjoint schedules can be run in any order, leading to the same configuration.

The events e and e' are disjoint since p' ≠ p

Which means we can do this:



$D_1 = e'(D_0)$, which should be impossible since they have different valencies.

So case 1 leads to a contradiction.

Case 2. p' = p

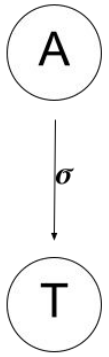
Recall: disjoint schedules can be run in any order, leading to the same configuration.

Consider any finite *deciding run* from A where p takes no steps.

Let T = σ(A)

Case 2. p' = p
Recall: disjoint schedules can be run in any order, leading to the same configuration.

Consider any finite *deciding run* from A where p takes no steps.
Let T = σ(A)


$$\begin{tikzcd} A \arrow[d, "\sigma"] \\ T \end{tikzcd}$$

Case 2. p' = p

Recall: disjoint schedules can be run in any order, leading to the same configuration.

Consider any finite *deciding run* from A where p takes no steps.

Let T = σ(A)

σ is also applicable to D_0

Looks good? T is still univalent.

$$\begin{tikzcd}
D_0 \arrow[d, "\sigma"] & A \arrow[l, "e"'] \arrow[d, "\sigma"] \\
E_0 & T \arrow[l, "e"']
\end{tikzcd}$$

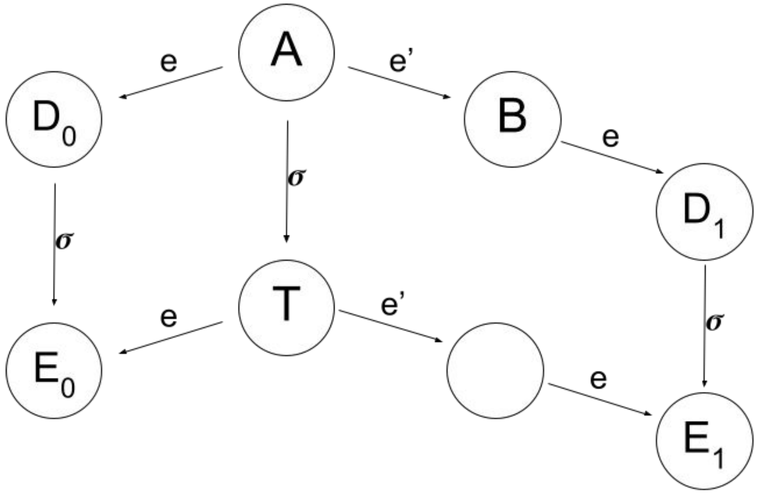
Case 2. p' = p
Recall: disjoint schedules can be run in any order, leading to the same configuration.

Consider any finite *deciding run* from A where p takes no steps.
Let T = σ(A)

σ is also applicable to D_0
Looks good? T is still univalent.

But σ is also applicable to D_1

T can reach E_0 and E_1 so T is bivalent.
But the run from A to T was a deciding run.
So <u>case 2 leads to a contradiction.</u>



Lemma 3.

In both case 1 and case 2, there was a contradiction. So our assumption that $\mathbb{D}$ contains no bivalent configurations was false.

Thus $\mathbb{D}$ does contain a bivalent configuration.

$\square$

Recap

From Lemma 2 we showed that there must be a bivalent initial configuration for consensus protocol P

From Lemma 3 we showed that in any bivalent configuration C, there is a bivalent configuration C' reachable from C by a schedule where some event e is the last event applied.

Admissible run:

1. All processes take infinitely many steps except the faulty process.
2. All messages sent to nonfaulty processes are eventually received.

Theorem 1.

We design a method of execution where we ensure every run is admissible.

The run is constructed in *stages*, starting from an initial configuration.
A queue of processes is maintained in an arbitrary order.
The message buffer is ordered according to the order the messages were sent, FIFO.

Admissible run:

1. All processes take infinitely many steps except the faulty process.
2. All messages sent to nonfaulty processes are eventually received.

Theorem 1.

A *stage* is one or more process steps.

The stage ends with the first process in the process queue taking a step in which, if its message queue was not empty at the start of the stage, its earliest message is received.

Admissible run:

1. All processes take infinitely many steps except the faulty process.
2. All messages sent to nonfaulty processes are eventually received.

Theorem 1.

A *stage* is one or more process steps.

The stage ends with the first process in the process queue taking a step in which, if its message queue was not empty at the start of the stage, its earliest message is received.

That process is then moved to the back of the process queue.
In an infinite sequence of *stages*, every process takes infinitely many steps, and receives every message sent to it.
Therefore the run is admissible.

Admissible run:

1. All processes take infinitely many steps except the faulty process.
2. All messages sent to nonfaulty processes are eventually received.

Theorem 1.

Let our initial configuration be bivalent as proven possible by Lemma 2.

Let our stage be defined by a schedule described in Lemma 3, where the last event applied is e.

Since e is the last event applied in each stage, the next stage can always begin in a bivalent configuration. This can continue indefinitely. The run is admissible and no result is ever achieved.

Thus, protocol P is not totally correct. $\square$

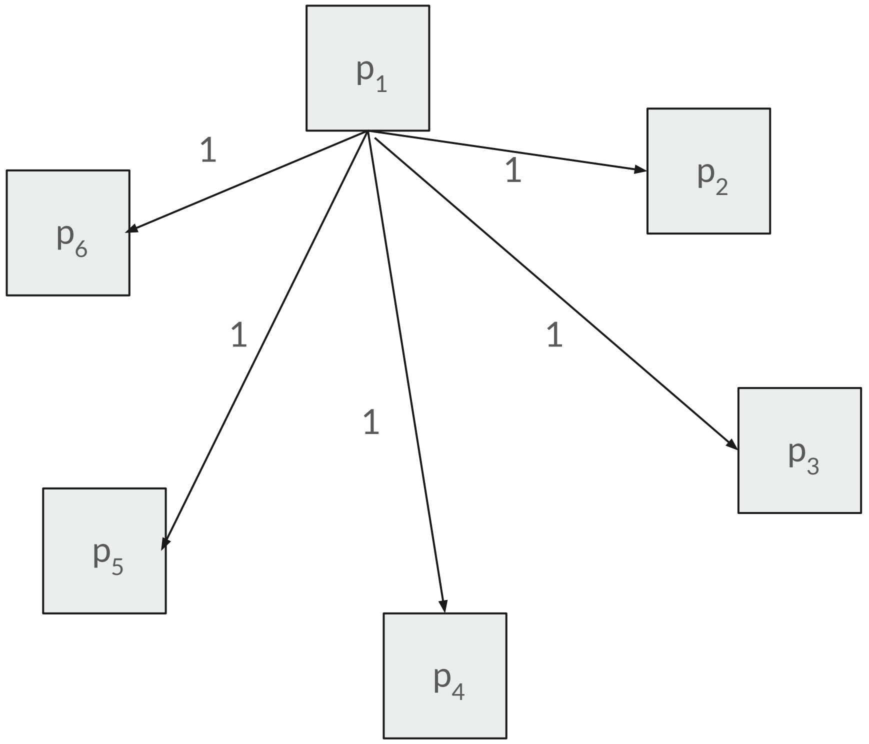
What if processes are initially dead?

Assume no process dies during the execution of the protocol and no process knows which processes are initially dead.

What if processes are initially dead?

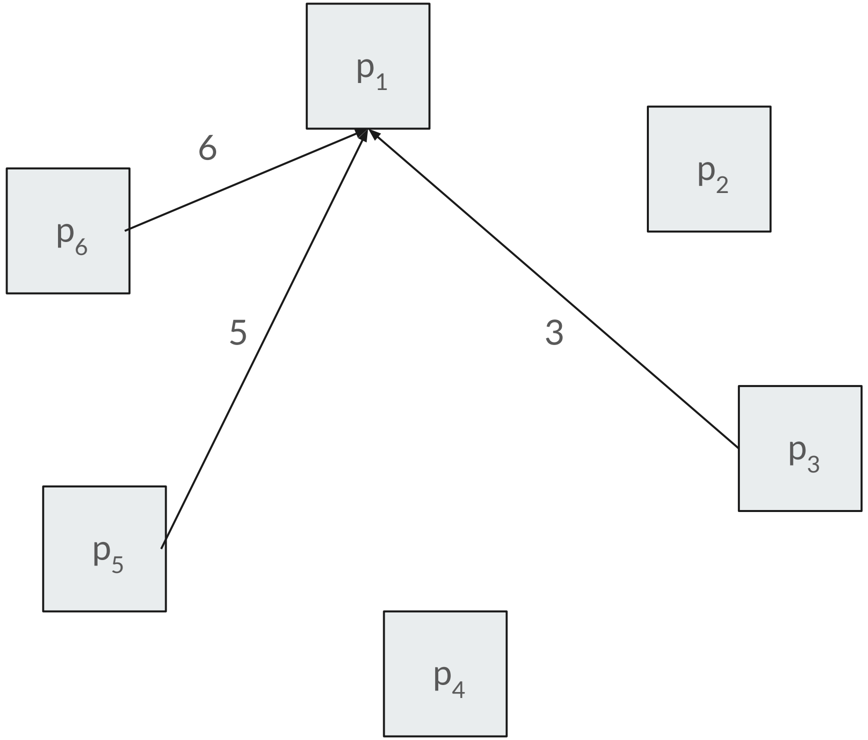
Assume no process dies during the execution of the protocol and no process knows which processes are initially dead.

If majority of the processes are nonfaulty, we can solve the consensus problem!



Let L = ceil((N + 1)/2).

1. Each process sends message to all other processes containing its process number.



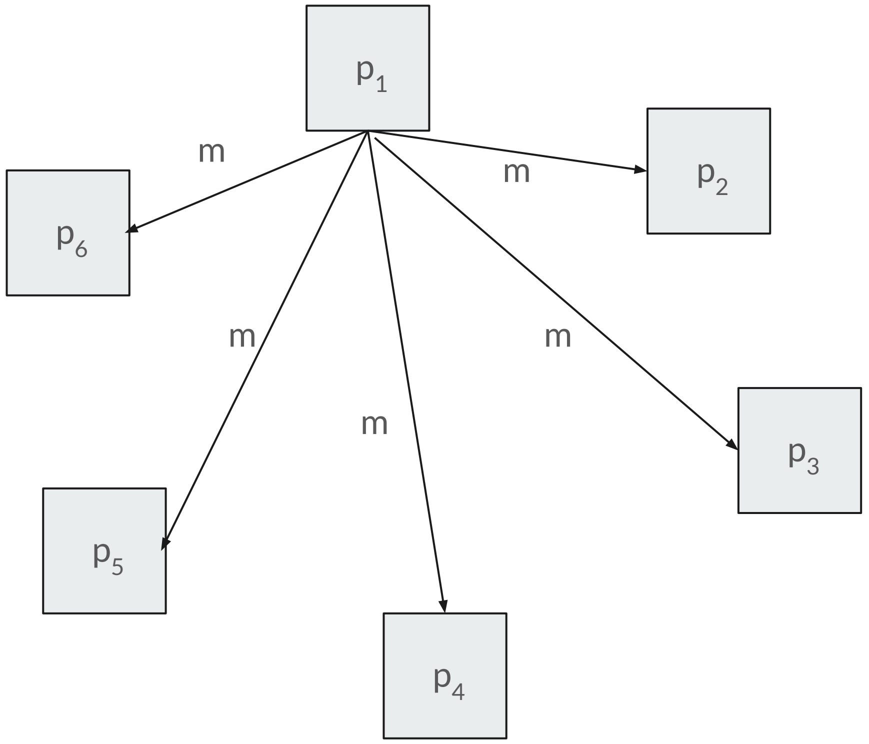
Let L = ceil((N + 1)/2).

1. Each process sends message to all other processes containing its process number.

2. Each process listens for messages from L-1 other processes. There must exist such a time since majority of processes are nonfaulty.

   p_1 hears from [p_3, p_5, p_6]

$m = (p_1, [p_3, p_5, p_6], init(p_1))$



Let L = ceil((N + 1)/2).

1. Each process sends message to all other processes containing its process number.

2. Each process listens for messages from L-1 other processes. There must exist such a time since majority of processes are nonfaulty.

   p_1 hears from $[p_3, p_5, p_6]$

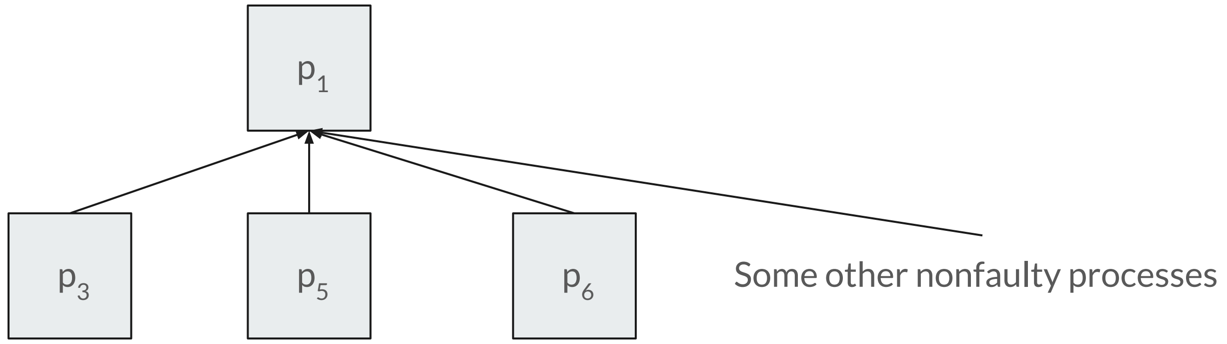
3. Each process sends message containing itself + processes they heard from in step 2 + its initial value

   p_1 sends out $(p_1, [p_3, p_5, p_6], init(p_1))$

Each process waits till it hears from each process it knows. It gains new knowledge from the list of processes in each message.

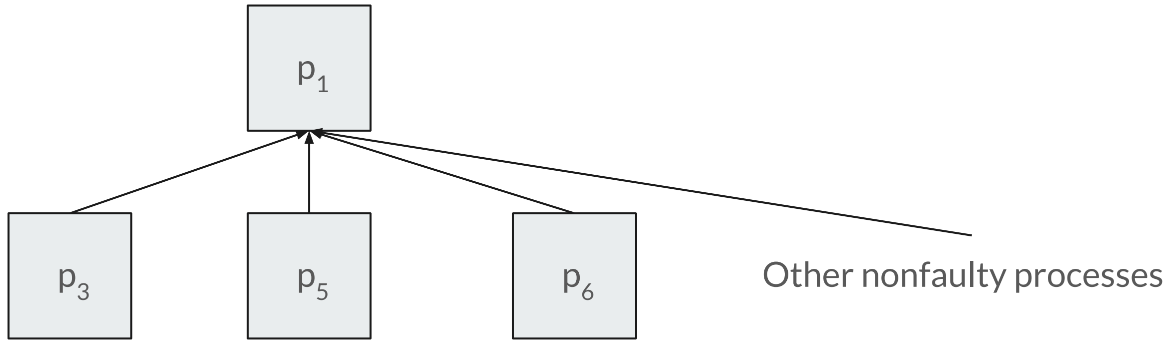
Each process waits till it hears from each process it knows. It gains new knowledge from the list of processes in each message.

For example, for p_1:



Each process waits till it hears from each process it knows. It gains new knowledge from the list of processes in each message.

For example, for p_1:



These processes form a complete graph of size at least L and only one can exist. Each process has access to the initial values of the other processes, so these processes can come to an agreement.