Impossibility Results: or why distributed systems are hard

Amy Babay

University of Pittsburgh
School of Computing and Information

Readings

- *CAP Twelve Years Later: How the "Rules" Have Changed*, Eric Brewer
- *Impossibility of Distributed Consensus with a Single Faulty Process,* Michael Fischer, Nancy Lynch, Michael Paterson

Why distribute?

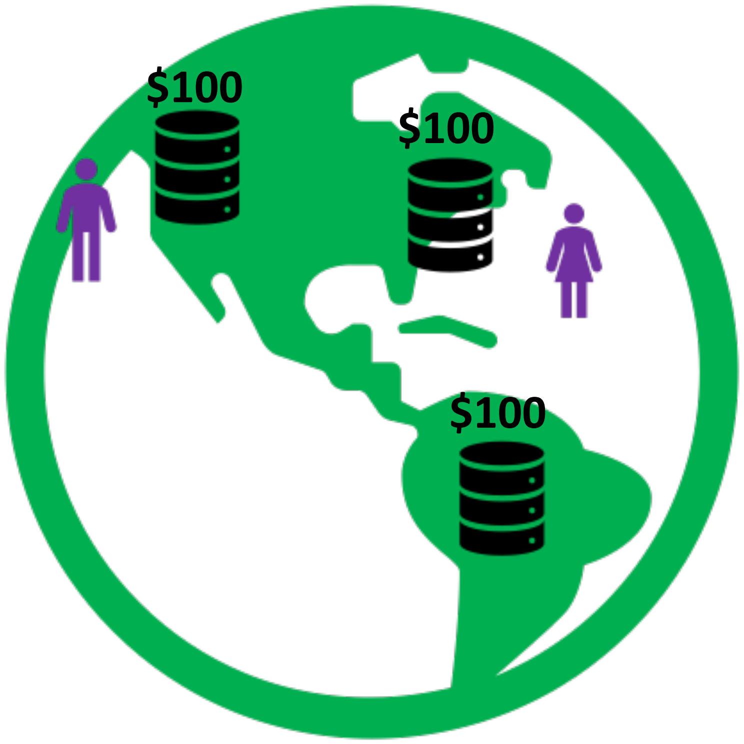
- Wouldn’t our lives be easier if we just had a single database/data store/control server/etc?

Why distribute?

- Availability
  - What if my single server fails?
- Scale
  - What if my data can’t all fit on one server?
- Performance
  - Latency: put data/services closer to geographically distributed users
  - Throughput: more servers can handle more requests
- Specialization
  - Systems may be composed of different types of components (e.g. sensors, storage servers, GPUs)

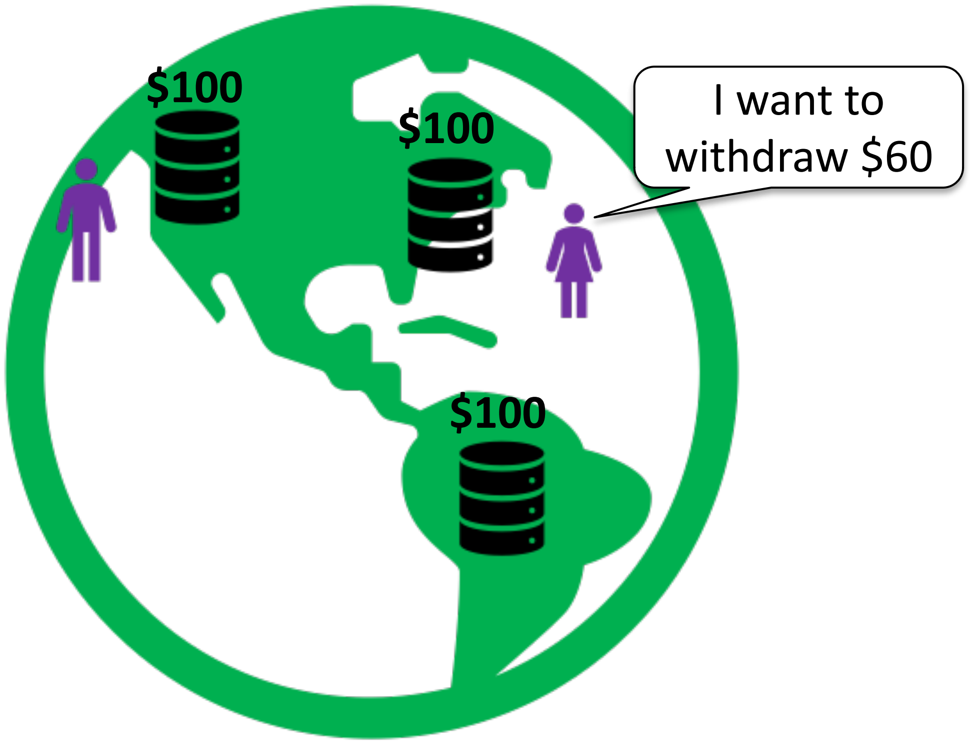
Sounds good…so what's the problem?

- Consider a database storing bank accounts (classic example)



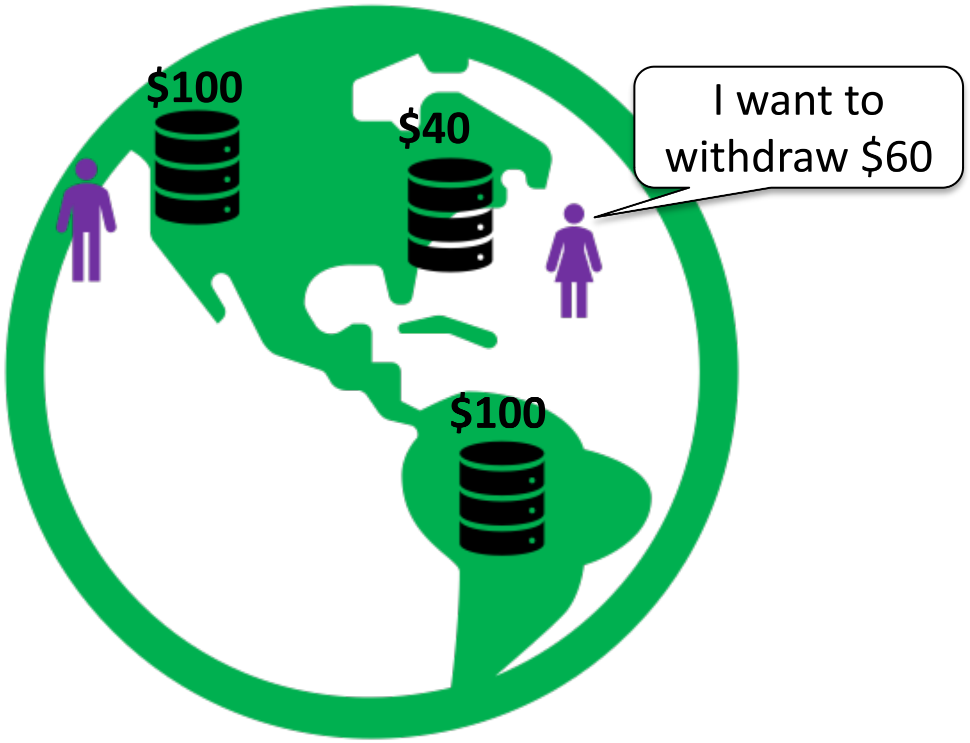
Sounds good…so what's the problem?

- Consider a database storing bank accounts (classic example)



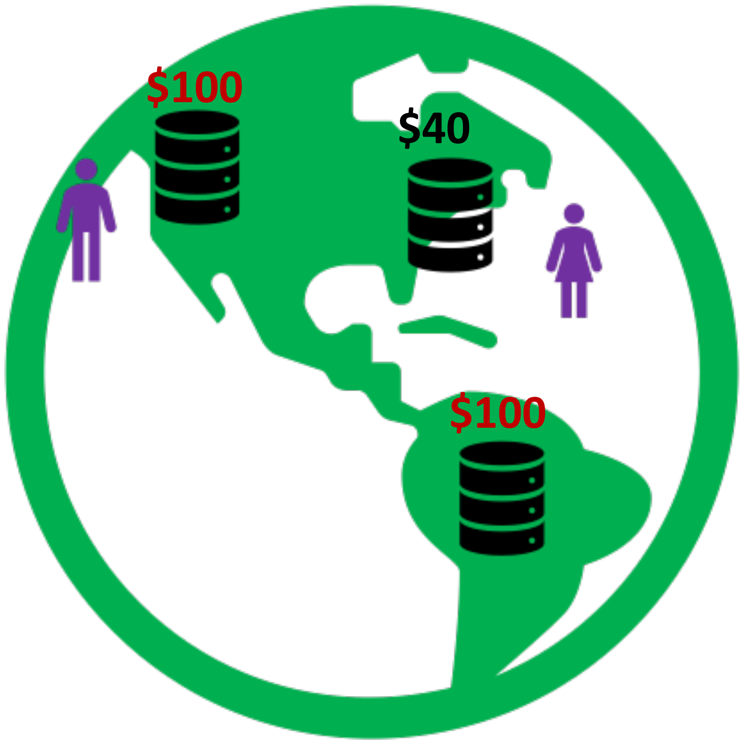
Sounds good…so what's the problem?

- Consider a database storing bank accounts (classic example)



Sounds good…so what's the problem?

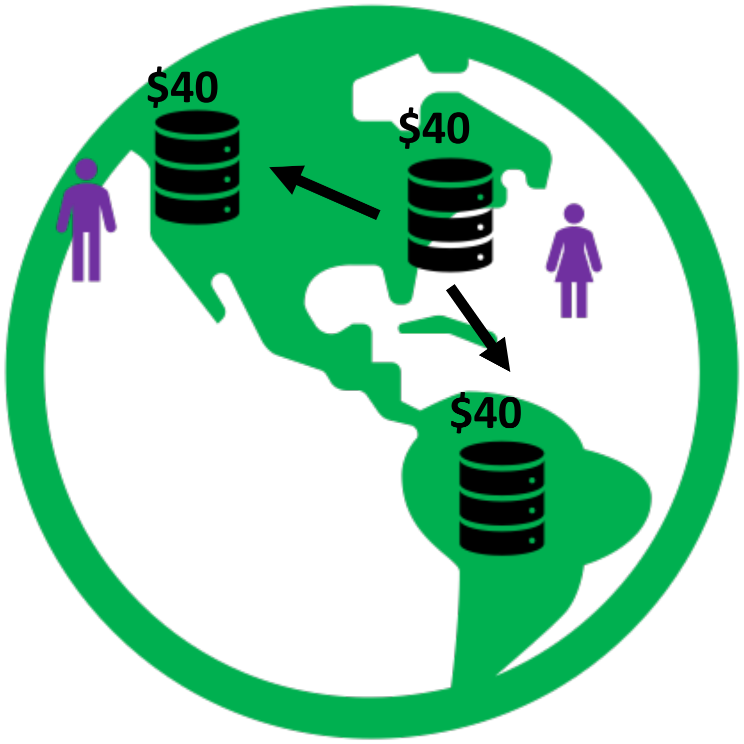
- Consider a database storing bank accounts (classic example)



Sounds good…so what's the problem?

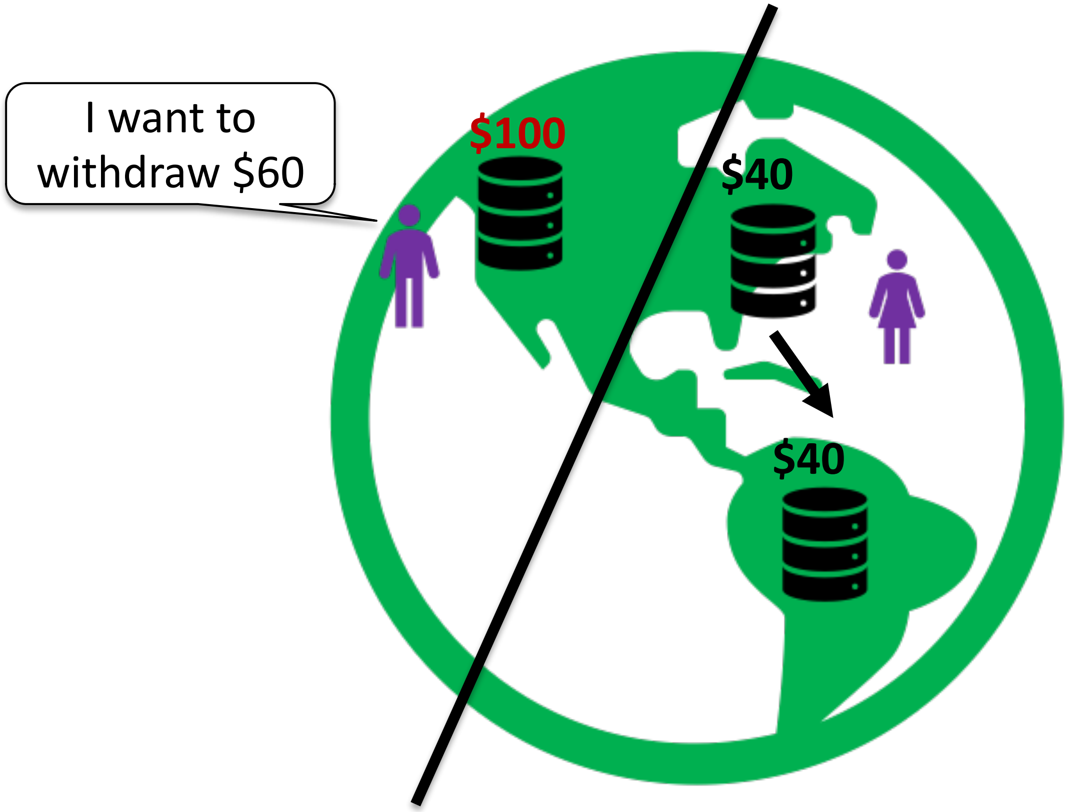
- Consider a database storing bank accounts (classic example)

We need a **synchronization** protocol



So we synchronize the accounts. Easy, right?

- When our replicas are **connected**, we can synchronize
- But, sometimes the network partitions….



With this partition, my server can't reach one of the others to update it

Our account doesn't really have $60 left to withdraw! What should happen?

CAP Theorem

- Introduced by Armando Fox and Eric Brewer in 1999 HotOS paper and Brewer's 2000 PODC Keynote
- C = Consistency
- A = Availability
- P = Partition Tolerance
- Theorem: Pick at most 2 – It is impossible to design a system that is always consistent and always available, given the possibility of partitions

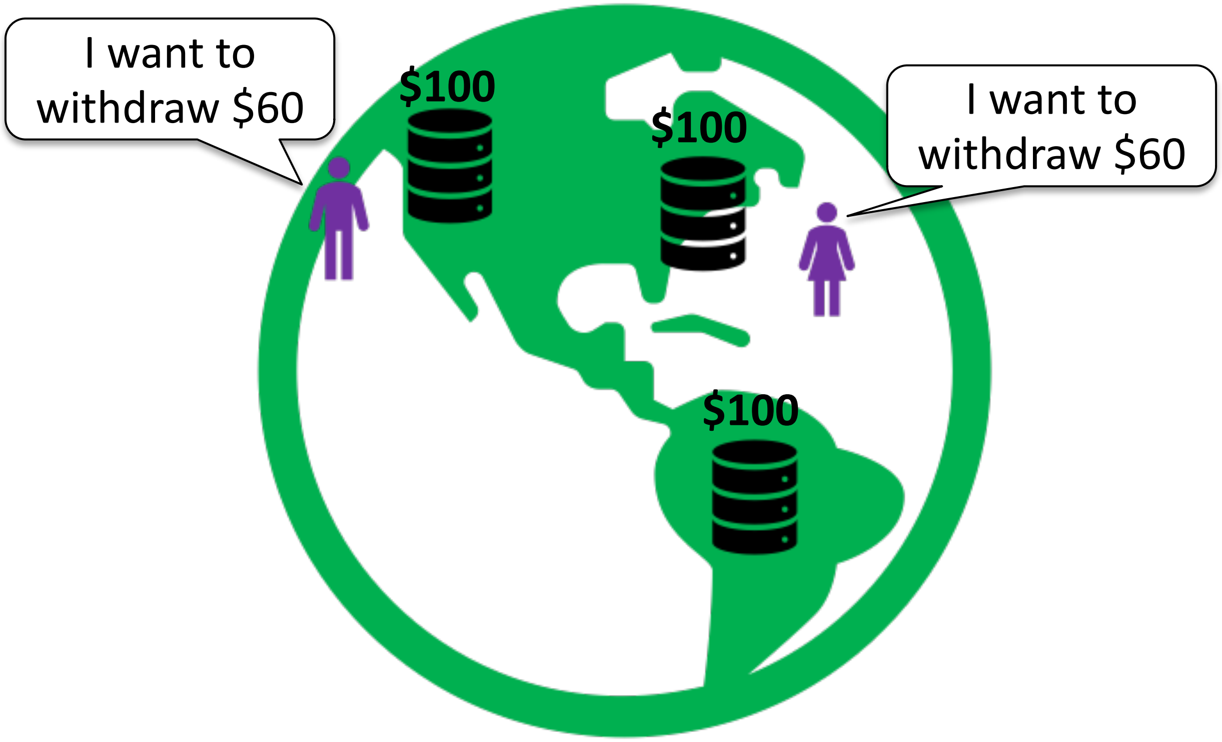
Consistency

- Single-copy consistency (serializability): observed behavior is the same as if there was a single server

Scenario 1 - OK
- Amy's withdrawal succeeds
- Chris's withdrawal fails
- Ending balance is $40

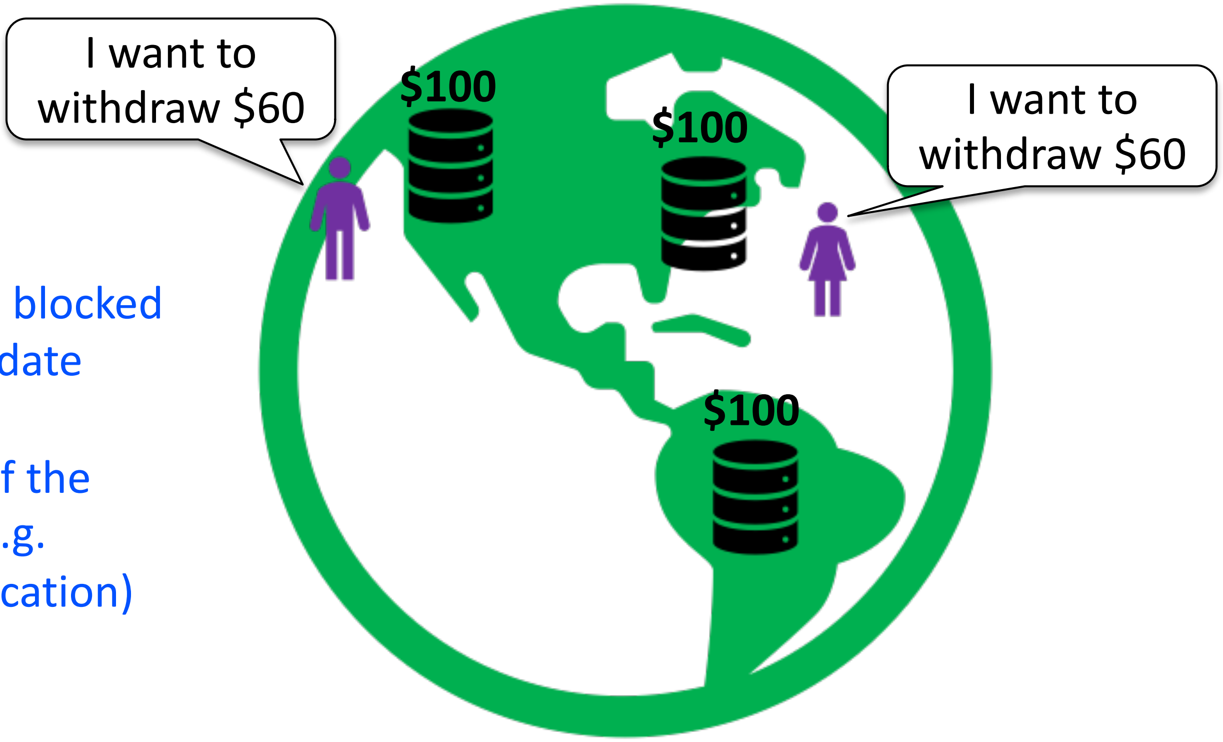
Scenario 2 - OK
- Chris's withdrawal succeeds
- Amy's withdrawal fails
- Ending balance is $40

Scenario 3 – NOT OK
- Amy's withdrawal succeeds
- Chris's withdrawal succeeds
- Ending balance is ???



Availability

- Users can perform operations (e.g. read and write data)

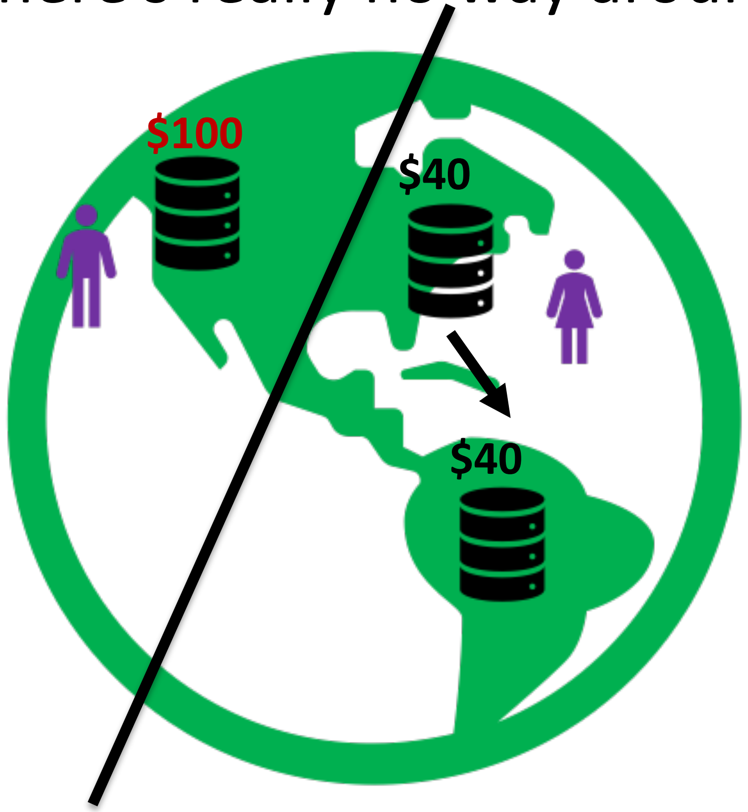


Neither person should be blocked from performing their update

(even though the result of the update can be a failure, e.g. insufficient balance notification)

Partition Tolerance

- Network partitions (disconnections) happen.
- We can do our best to make them as infrequent as possible, but there's really no way around this



So, “2 of 3” is misleading

- You don’t really get to pick CA…
- The world doesn’t end when a partition happens, so your system has to do **something** in that case
  - Although that may be to sacrifice availability and stop accepting updates

- But, most of the time the network isn’t partitioned!
- So consistency and availability are both possible in the **normal case**

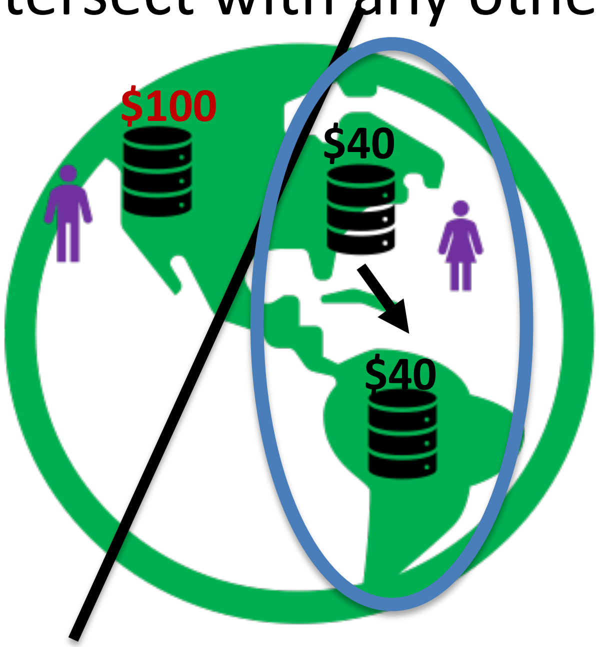
- And, it’s not a binary choice
- There is a whole **range of trade-offs** we can make between Consistency and Availability

What should we do during partitions?

- Detect and manage
  - How we manage depends on trade-off we want between consistency and availability
- Note: detection isn’t completely straightforward
  - Brings up classic distributed systems issues: slow vs. disconnected/failed, different nodes can have different views

Options for Managing Partitions

- Strongly consistent option
  - Only allow operations to proceed in the part of the system with a **quorum** (if one exists)
  - Quorum: in distributed systems, this usually means a set that must intersect with any other set (e.g. majority)



Options for Managing Partitions

- More available approach – allow at least some operations to continue
  - Risky operations can be delayed
    - e.g. large withdrawal in banking
  - Safer operations can be applied
    - e.g. deposit in banking
  - Mistakes can be compensated for after recovery
- "Risky" and "Safe" operations strongly depend on the details of the specific system

Partition Recovery

- Generally requires that nodes keep a history during partition
- **Possible recovery approach**: Impose a total order on the operation history, and replay from the beginning of the partition
  - Must maintain order for **causally related** operations. Non-causally related operation may conflict

Resolving Conflicts

- Approach 1: avoid conflicts
  - Only allow **commutative** operations, so execution order doesn't matter
  - Have some well-defined merge/convergence procedure (shopping cart example)
- Approach 2: fix mistakes
  - Simple: "last writer wins"
  - Compensating transaction: undo the effects

Impossibility of Distributed Consensus with One Faulty Process (FLP result)

A closer look at synchronization

Consensus

- A group of processes needs to decide on a single value
- Every process may propose a value, and exactly one should be chosen
  - In systems that use a strongly consistent replication approach, this can be used to impose an **ordering** on submitted updates (state-machine replication). e.g. agree on the first update, then the second, etc.

Requirements for Consensus

- **Agreement**: all correct processes decide on the **same value**
- **Validity**: If a correct process decides on a value, that value was proposed by some process
- **Termination**: all correct processes eventually decide

Simple Asynchronous Model

- Each process starts with an initial input value in {0, 1}
- Processes communicate by sending messages to each other
- Communication is reliable, but asynchronous
  - Every message sent is **eventually received**
  - No bound on how long it takes to receive a message
- Correct (nonfaulty) processes run forever ("take infinitely many steps")
  - step = receive message, update state (based on message), send messages
- Faulty processes stop executing at some point (fail-stop)

FLP: Requirements for Consensus

- **Agreement**: all correct processes that decide on a value decide on the **same value**
  - "No accessible configuration has more than one decision value"
- **Validity**: 0 and 1 are both possible decision values
  - "For each $v \in \{0, 1\}$, some accessible configuration has decision value *v*"
  - Weaker requirement than typical (makes result stronger)
- **Termination**: Some correct process eventually decides
  - Weaker requirement than typical (makes result stronger)

Impossibility Result

- *No consensus protocol* is totally correct (meets requirements on previous slide) in spite of one fault

- What if there are no faults? How would you solve this problem?
- Why doesn't that work if a process **may** fail?

Impossibility Proof: Definitions

- Definitions:
  - A configuration is **0-valent** if 0 is the only reachable decision value
  - A configuration is **1-valent** if 1 is the only reachable decision value
  - A configuration is **bivalent** if both 0 and 1 are reachable decision values

Impossibility Proof: Sketch

- **Lemma 1** (Lemma 2 in paper): Any consensus protocol must have a bivalent initial configuration
- **Lemma 2** (Lemma 3 in paper): For any bivalent configuration and any pending message, we can take some sequence of steps in the protocol that ends with us applying that message and ending up in another bivalent configuration

- Thus, we can keep taking steps forever and never reach a decision (univalent configuration)

Impossibility Proof: Lemma 1

- *Lemma 1: Any consensus protocol that is totally correct in spite of one fault must have a bivalent initial configuration*

Proof: Lemma 1

- Assume the contrary…
- By definition, there must be a 0-valent initial configuration and a 1-valent initial configuration
- Consider initial values (0, 0, …, 0) -> 0 decision
- Consider initial values (1, 1, …, 1) -> 1 decision
- There is a chain of configurations
  - (0, 0, …, 0), (1, 0, …, 0), (1, 1, …, 0), … (1, 1, …, 1)
- At some point, there must be a pair of configurations C_0 & C_1 such that C_0 is 0-valent and C_1 is 1-valent, and their only difference is the starting value of one processor *p*

Proof: Lemma 1

- There must be a pair of configurations C_0 & C_1 such that C_0 is 0-valent and C_1 is 1-valent, and their only difference is the starting value of one processor *p*
- Let processor *p* immediately fail
  - e.g. (0,0,1), (0,1,1) -> (0,X,1), (0,X,1)
- C_0 & C_1 must reach the same decision
  - They only differed in *p*'s initial value
  - So, if *p* never does anything, they look exactly the same
- => either C_0 or C_1 must be a bivalent configuration

Lemma 2

- Lemma 2: Let *C* be a bivalent configuration of *P*, and let $e = (p, m)$ be an event that is applicable to *C*. Let $\mathcal{C}$ be the set of configurations of *C* without applying *e*, and let $\mathcal{D} = e(\mathcal{C}) = \{e(E) | E \in \mathcal{C} \text{ and } e \text{ is applicable to } E\}$. Then, $\mathcal{D}$ contains a bivalent configuration.

- Informally, for any bivalent configuration *C* and pending event *e*, there is some sequence of events that ends with event *e* and brings us to another bivalent configuration

Proof Intuition: Lemma 2

- Assume the contrary: assume that after some event $e = (p,m)$ we go from a bivalent configuration to **only univalent configurations** being reachable
  - $e = (p,m)$: event *e,* where process *p* receives message *m*

- Since the initial configuration is bivalent, both 0-valent and 1-valent configurations must be reachable
  - So there must be two "neighboring" configurations C_0 and C_1 that differ in only a single event *e'* at process p, which occurs (C_1) or doesn't occur (C_0) before "decision point" e

Proof Intuition: Lemma 2

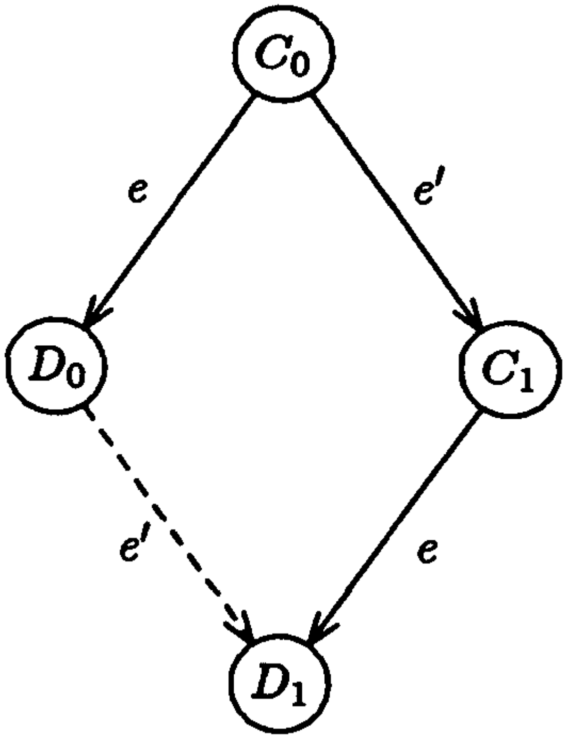


FIGURE 2

- C_0 is bivalent. Receiving e' then e commits to 1-valent execution; receiving e without having received e' commits to 0–valent execution

Proof Intuition: Lemma 2

- A series of steps σ that doesn't involve process *p* can't affect p's decision, so the ordering of σ vs *e* and *e'* doesn't change the decision
- But our protocol must be fault-tolerant – so what happens if *p* crashes?
  - The other processes must still decide on **some** value!

Proof Intuition: Lemma 2

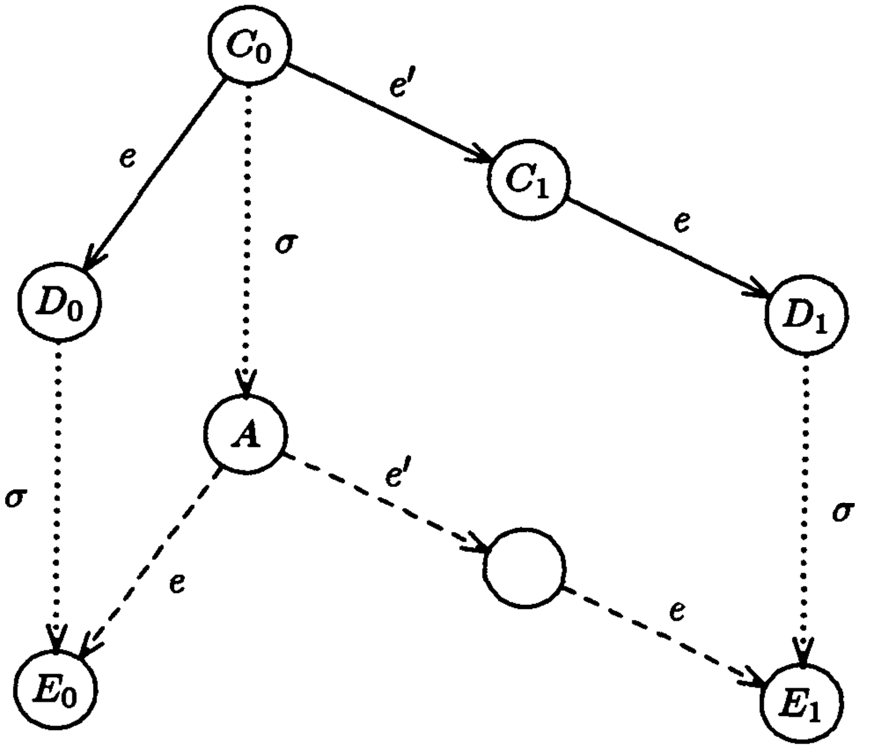


FIGURE 3

- The execution ending in state **A** (*p* fails after C_0) must decide 0 or 1
- **But what if *p* was only slow??**

Implications of FLP

- Why do we care?
- Strong, fundamental result
- But, if consensus is impossible, why are we studying consensus protocols next week?
  - What can we give up in the "Requirements for Consensus" to build a practical system?
- Discussion: How does CAP relate to FLP?