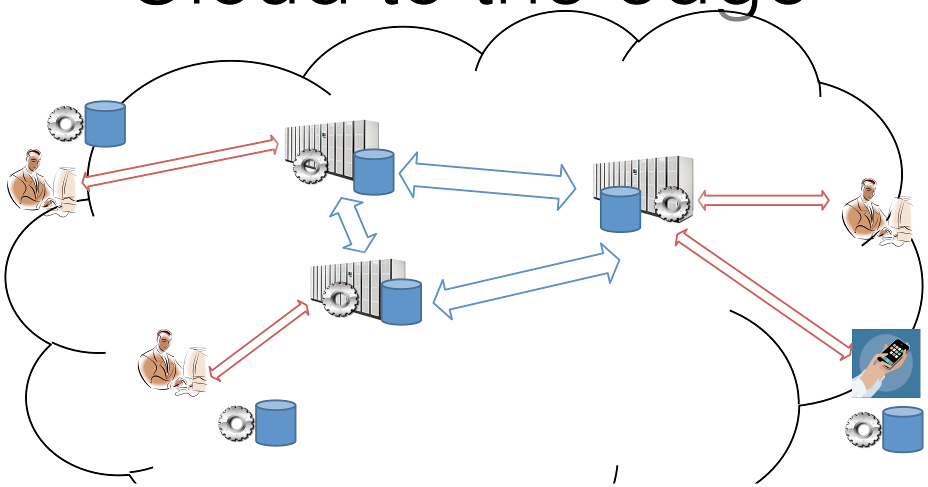
Just-Right Consistency

Static analysis for minimal synchronisation

Marc Shapiro

Inria &
Sorbonne-Universités UPMC-LIP6

Cloud to the edge



Social, web, e-commerce: shared mutable data

Scalability ⇒ replication ⇒ consistency issues

Part I: **Consistency vs. performance**

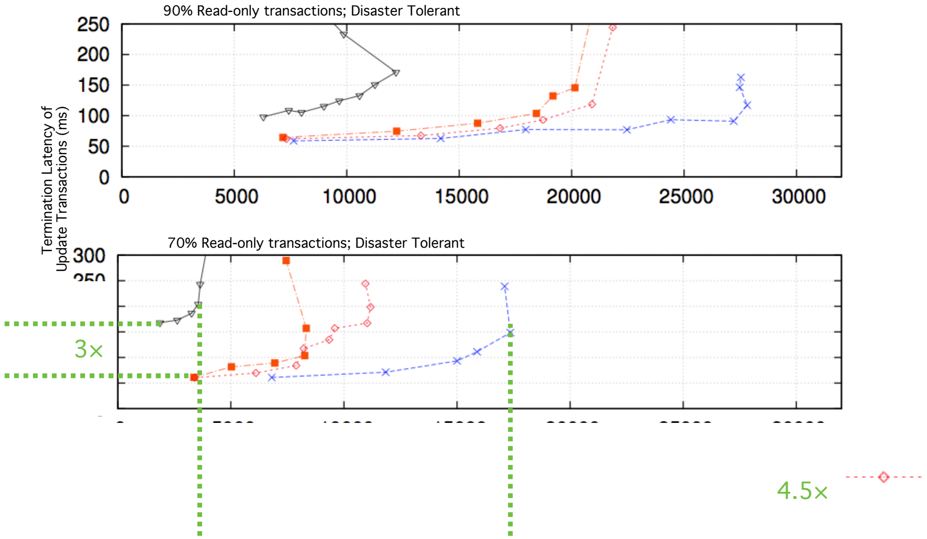
Part I: Consistency vs. performance

- Geo-replicated cloud databases
- Consistency models
- Some partial solutions

Part II

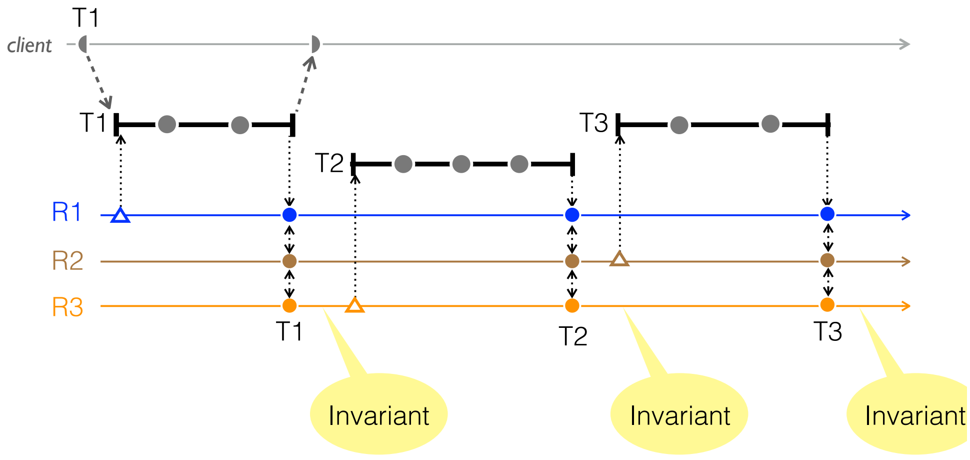
- Just-right consistency

Models matter

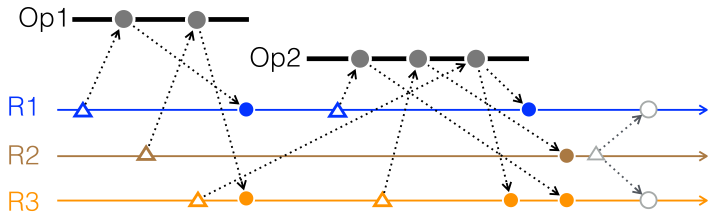


Credit: Masoud Saeida Ardekani

Strongest: Strict Serialisability



Weakest: Eventual consistency



The problem(s) of consistency

Same object:

- *Safe:* updates, state satisfy specification, internal invariants
- Replicas *converge* to same state

Separate objects: maintain relations

- *Multi-object invariants*
- Different kinds ⟹ different mechanisms

ACID transactions mix all this; often too strong

Seq. consistency examples

Bank account

- *debit(amt), credit(amt), accrueInterest(amt)*
- Invariant: *“balance ≥ 0”*
- { *amt ≤ balance* ∧ *Inv* } ***debit(amt)*** { *Inv* }

File system

- *mkdir, rmdir, create, write, rm, ls,* etc.
- Invariant: Tree
- { Tree ∧ ¬ *x/…/y* } ***mv(x,y)*** { Tree }

CAP

Sequential Consistency: total order of operations ⟹ replicas identical

- Consensus: *“Who’s next?”*
- Requires communication

CAP Theorem: “When network can **P**artition,

- either sequential **C**onsistency,
- or **A**vailability;
- can’t have both!”

Availability related to performance

- Parallelise
- More implementation choices

Consistency issues under EC

Updates delivered in different orders: not identical, do not converge

Lost updates (LWW: by design)

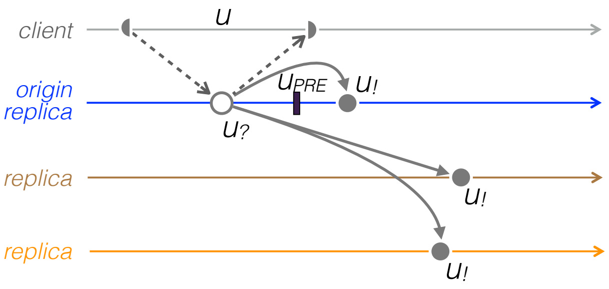
No causality: updates received out of order

No transactions: inter-object invariants violated

Compensating at application level: very challenging

Solution: Spanner?

Operation



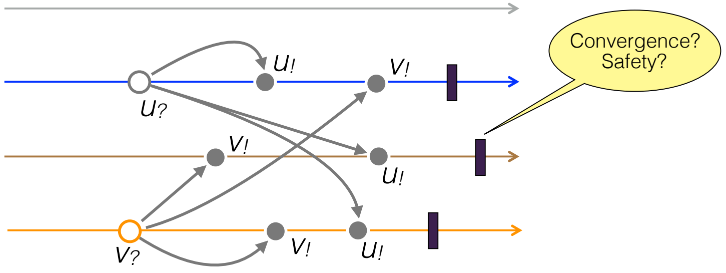
u: state ⤳ (retval, (state ⤳ state))

Prepare (@origin) $u_?$; deliver $u_!$

Read one, write all (ROWA)

Deferred-update replication (DUR)

Concurrency



Concurrent, Multi-master

Strong: total order, identical state

Weak: concurrent, interleaving, no global state

Anomalies of concurrent updates

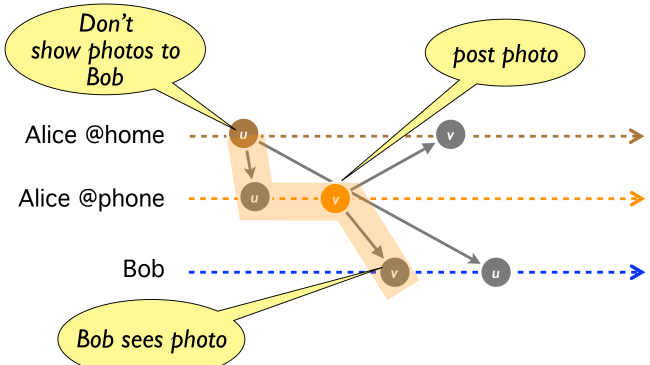
Bank:

- σ_{init} = 100€
- Alice: *credit(20)* = { σ ≔ 120 }
- Bob: *debit (60)* = { σ ≔ 40 }
- σ = ???

File system:

- σ_{init} = "/"
- Alice: *mkdir ("/foo"); mkdir ("/foo/bar")*
- Bob: receives *mkdir ("/foo/bar")*
- σ = ???

Not causal



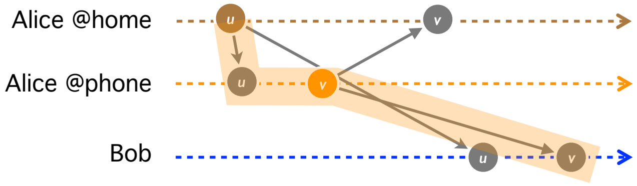
access (Bob, photo) ⟹ *ACL (Bob, photo)*

v observed effects of *u*

⟹ *v* should be delivered after *u*

Available: doesn't slow down sender

(1) Causal consistency



access (Bob, photo) ⟹ *ACL (Bob, photo)*

v observed effects of *u*

⟹ *v* should be delivered after *u*

Available: doesn't slow down sender

(2) Conflict-free replicated data types

Data type

- Encapsulates issues

Replicated

- At multiple nodes

Available

- Update my replica without coordination
- Convergence guaranteed (formal properties)
- Decentralised, peer-to-peer

Commute ⟹ converge

Bank account:

- ***credit(amt)***$_!$ = { *local_balance += amt* }
- ***debit(amt)***$_!$ = { *local_balance –= amt* }
- ***interest()***$_!$ =
  { *local_balance += origin_balance**.05 }

File system:

- *write(f)*$_!$ = { *local_f* ⊔ *f* }

CRDT design concept

Backward-compatible with sequential datatype

Commute ⟹ concurrent is same

- *add(e); rm(f) = rm(f); add(e)* ≜ *add(e)* || *rm (f)*

Otherwise, *concurrency semantics*

- Example: *add(e)* || *rm (e)*
- Deterministic, similar to sequential
  - ≈ *rm(e);add(e)* or ≈ *add(e); rm(e)*
- Merge, don't lose updates
- Result doesn't depend on order received
- Stable preconditions

CRDT design concept

Backward-compatible with sequential datatype

Commute ⟹ concurrent is same

- *add(e); rm(f) = rm(f); add(e)* ≜ *add(e)* || *rm (f)*

Otherwise, *concurrency semantics*

- Example: *add(e)* || *rm (e)*
- Deterministic, similar to sequential
  - ≈ *rm(e);add(e)* or ≈ *add(e); rm(e)*
- Merge, don't lose updates
- Result doesn't depend on order received
- Stable preconditions

CRDT design concept

Backward-compatible with sequential datatype

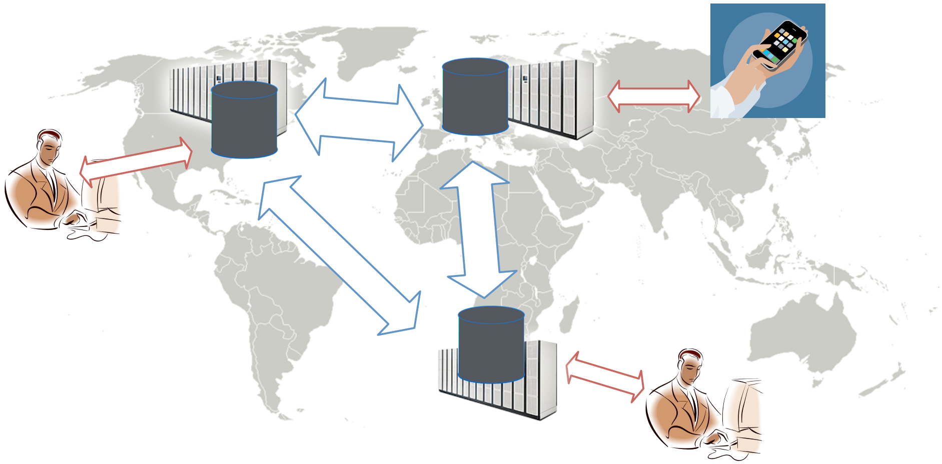
Commute ⟹ concurrent is same

- *add(e); rm(f) = rm(f); add(e)* ≜ *add(e)* || *rm (f)*

Otherwise, *concurrency semantics*

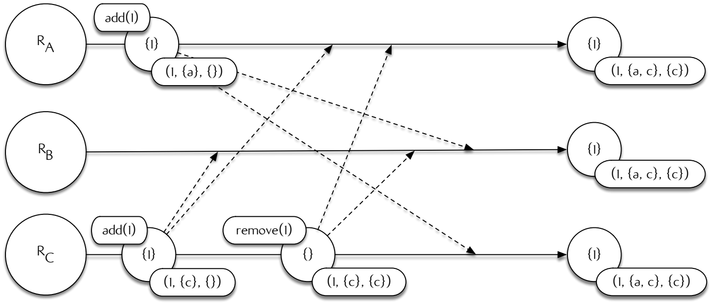
- Example: *add(e)* || *rm (e)*
- Deterministic, similar to sequential
  - ≈ *rm(e);add(e)* or ≈ *add(e); rm(e)*
- Merge, don't lose updates
- Result doesn't depend on order received
- Stable preconditions

CRDT concept



[Just-Right Consistency]

Add-Wins Set CRDT



[CRDTs in practice]

CRDT types

Converge concurrent updates
Encapsulate replication & resolution
Re-usable data types
Correct by construction

Register
- Last-Writer Wins
- Multi-Value

Set
- Grow-Only
- 2P
- Observed-Remove

Map

Counter
- Unlimited
- Restricted ≥0

Graph
- Directed
- Monotonic DAG
- Edit graph

Sequence

[Just-Right Consistency]

(3) Bounded Counter CRDT

Replicated Counter: ***inc(), dec()***

Invariant: bounded "$x \geq 0$"

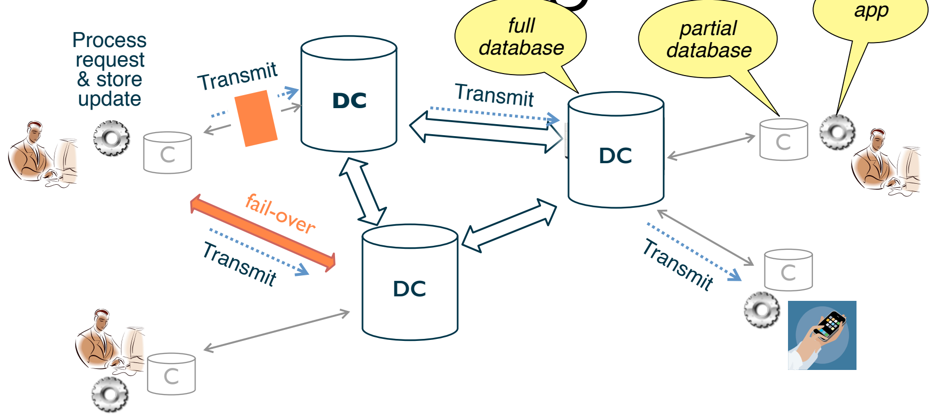
Credit per replica: $\sum credit_i \leq bound$

Asynchronous:
- $\{\ credit_i \geq 1\ \}$ ***dec***$_{!}$***()*** $=\{\ ctr -= 1;\ credit_i -= 1\ \}$
- ***transfer (credit***$_i$***, credit***$_i$***)***

Synchronized
- ***acquire(credit***$_i$***)***

[Just-Right Consistency]

SwiftCloud edge +cloud



Update, commit shared store locally
Availability + consistency: DC switch
Causal + transactional
3000+ client replicas

Antidote

SyncFree EU project

High performance, sharded, transactional, causal

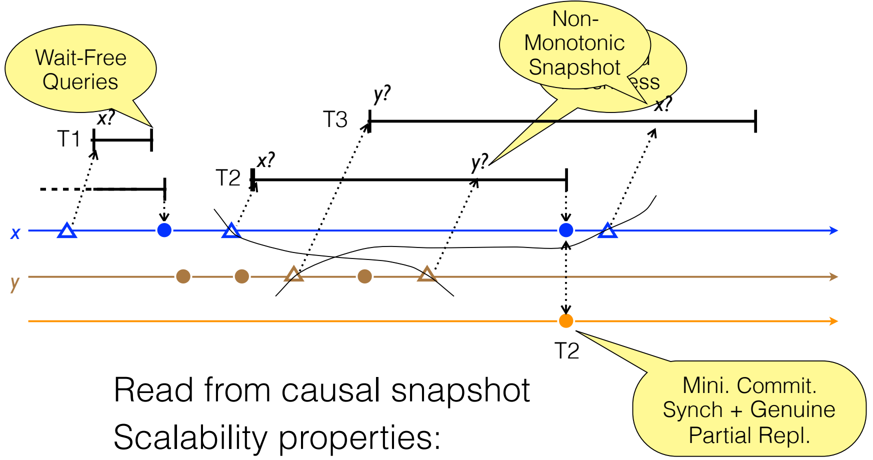
Aims to scale to 100s of DCs

- Very modular
- Partial replication
- Small but safe metadata (vector clock)

In DC: strong consistency, physical clocks (Clock-SI)

Industrial apps: Virtual Wallet, SocialApp, configuration management, FMK

(4) NMSI: strong, parallel

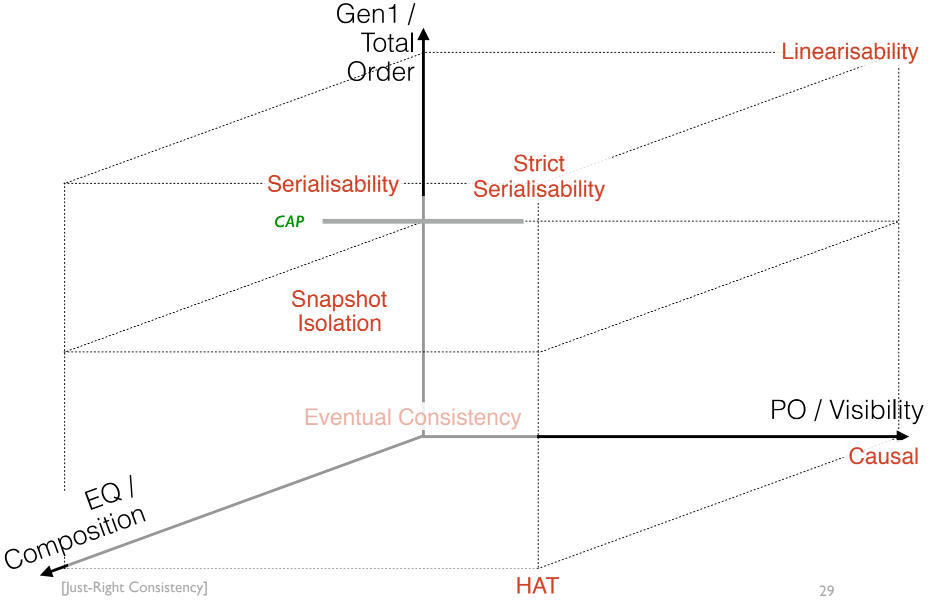


Read from causal snapshot

Scalability properties:

- Wait-Free Queries
- Forward Freshness
- Mini. Commitment Synchronisation
- Genuine Partial Replication

Three dimensions



Part II: **Just-right consistency**

Part I: Consistency vs. performance

- Geo-replicated cloud databases
- Consistency models
- Some partial solutions

Part II

- Just-right consistency

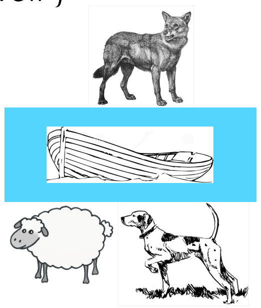
Application invariants

$South \uplus Boat \uplus North = \{ sheep, dog, wolf \}$

$carryNorth(S) \implies 1 \leq |S| \leq 2$

$carrySouth(S) \implies 1 \leq |S| \leq 2$

$\forall S \in \{South, Boat, North\} :$
$\quad sheep \in S \land wolf \in S \implies dog \in S$



Hard to tease invariants out

- Silent invariants

Seq. consistency examples

Bank account

- *debit(amt), credit(amt), accrueInterest(amt)*
- Invariant: *"balance ≥ 0"*
- { *amt ≤ balance* ∧ *Inv* } *debit(amt)* { *Inv* }

File system

- *mkdir, rmdir, create, write, rm, ls*, etc.
- Invariant: Tree
- { Tree ∧ ¬ *x/…/y* } *mv(x,y)* { Tree }

Just-Right Consistency

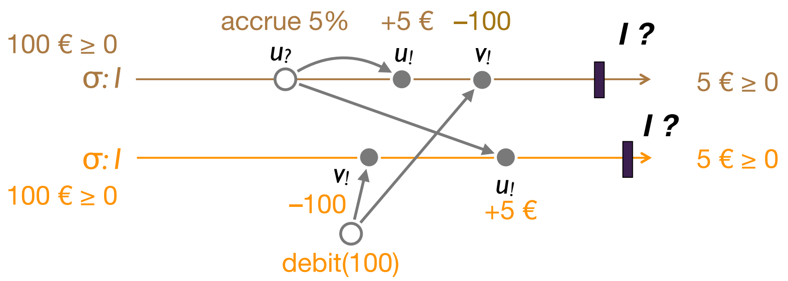
CRDT geo-replicated database

- Lots of internal parallelism
- Transactional, causal consistency by default

Specification of application updates, invariant

- CISE: do all state transitions preserve invariant?
- If not, fix: adjust
  - either specification
  - or synchronisation
- Repeat until safe

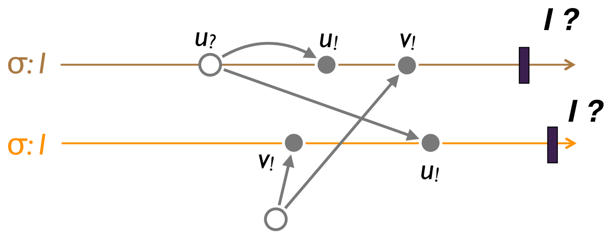
App / synch co-design: Minimal synchronisation



Asynchronous, replicated updates

- State σ
- Invariant I
- Prepare: read one, generate effector
- Update all, deferred: deliver effector

Converge? Invariant OK?



CISE Rules

1: Sequential correctness

- Any single operation maintains the invariant

2: Convergence

- Concurrent effectors commute

3: Precondition Stability

- Every precondition is stable under every concurrent operation

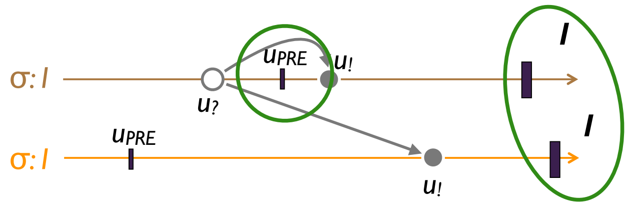
If satisfied: invariant is guaranteed

Simple example: bank account

Operations: *credit(amount), debit(amount)*

Invariant: *balance* ≥ *0*

- Start with weak specification
- Rule 1 ⟶ strengthen precondition for debit
- Rule 2: OK
- Rule 3 ⟶ *debit* || *debit* unsafe, fixed with concurrency control



CISE Rules

1: Sequential correctness

- Any single operation maintains the invariant

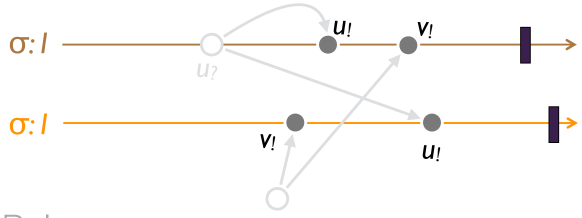
2: Convergence

- Concurrent effectors commute

3: Precondition Stability

- Every precondition is stable under every concurrent operation

If satisfied: invariant is guaranteed



CISE Rules

1: Sequential correctness

- Any single operation maintains the invariant

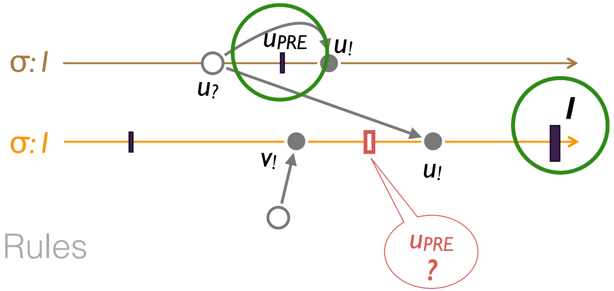
2: Convergence

- Concurrent effectors commute

3: Precondition Stability

- Every precondition is stable under every concurrent operation

If satisfied: invariant is guaranteed



CISE Rules

1: Sequential correctness

- Any single operation maintains the invariant

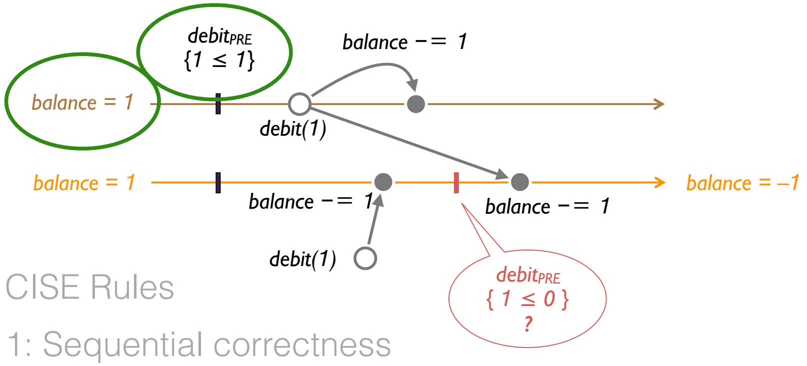
2: Convergence

- Concurrent effectors commute

3: Precondition Stability

- Every precondition is stable under every concurrent operation

If satisfied: invariant is guaranteed



CISE Rules

1: Sequential correctness

- Any single operation maintains the invariant

2: Convergence

- Concurrent effectors commute

Fix: concurrency control

3: Precondition Stability

- Every precondition is stable under every concurrent operation

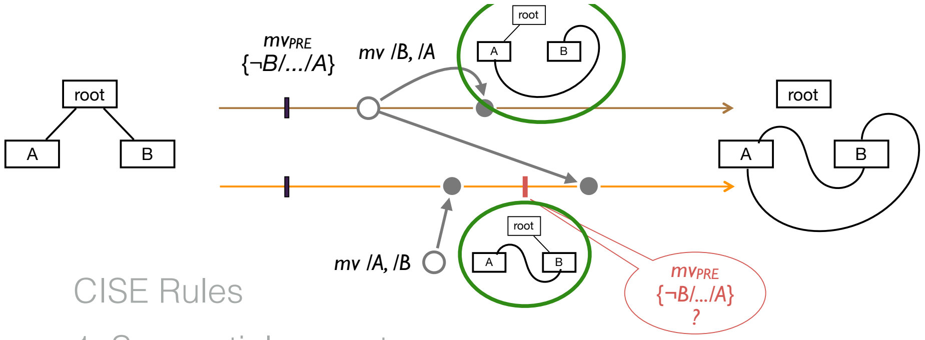
If satisfied: invariant is guaranteed

Advanced example: file system

Operations: *mkdir, rmdir*, *mv*, *update*, etc.

Invariant: Tree

- Rule 1 ⟶ precondition on *mv*
  *"May not **move** node under self"*
- Rule 2 ⟶ Use CRDTs for ***update*** || ***update***
- Rule 3 ⟶ ***mv*** || ***mv*** precondition unstable



CISE Rules

1: Sequential correctness

- Any single operation maintains the

2: Convergence

- Concurrent effectors commute

3: Precondition Stability

- Every precondition is stable under every concurrent operation

If satisfied: invariant is guaranteed

You can have your cake and eat it too

Applying the logic

Only *O*(n^2): no need to consider all possible interleavings

We use a tool

- You can apply the same logic manually

Conclusion & future work

3-D decomposition

- Deconstruct hierarchy
- Classes of invariants / primitive mechanisms

CISE tool

- Synthesize synchronisation

CISE assumes causal, transactional

- Constructive: use insights for designing apps, building mechanisms
- Deconstruct / weaker / chopping transactions
- Selective application of causality